Real-Time Systems Compilation

Dumitru Potop-Butucaru

Université Pierre et Marie Curie

Mémoire d'habilitation à diriger les recherches

Spécialité Informatique
École Doctorale Informatique, Télécommunications, et Électronique
(EDITE)

Compilation de systèmes temps réel
(Real-Time Systems Compilation)

par Dumitru Potop Butucaru

Présenté aux rapporteurs :

Sanjoy Baruah – Professeur, University of North Carolina
Nicolas Halbwachs – Directeur de recherche, CNRS/Vérimag
Reinhard von Hanxleden – Professeur, Université de Kiel

afin d'être soutenu publiquement le 10 novembre 2015 devant la commission d'examen formée de :

Albert Cohen – Directeur de recherche, INRIA Paris-Rocquencourt
Nicolas Halbwachs – Directeur de recherche, CNRS/Vérimag
Reinhard von Hanxleden – Professeur, Université de Kiel
François Irigoin – Directeur de recherche, MINES ParisTech
Alix Munier-Kordon – Professeur, Université Pierre et Marie Curie
François Pêcheux – Professeur, Université Pierre et Marie Curie
Renaud Sirdey – Directeur de recherche, CEA Saclay

Contents

1 Introduction **3**
- 1.1 When it all began... . . . 3
- 1.2 Overview of previous work . . . 5
- 1.3 Research project: Real-time systems compilation . . . 7

2 Introduction to synchronous languages **11**
- 2.1 Synchronous languages . . . 15
- 2.2 Related formalisms . . . 18

3 Automatic synthesis of optimal synchronization protocols **21**
- 3.1 Semantics of a simple example . . . 23
- 3.2 Problem definition . . . 24
  - 3.2.1 Previous work . . . 26
- 3.3 Contribution . . . 27
  - 3.3.1 Definition of weak endochrony . . . 27
  - 3.3.2 Characterization of delay-insensitive synchronous components . . . 28
  - 3.3.3 Synthesis of delay-insensitive concurrent implementations 29

4 Reconciling performance and predictability on a many-core **31**
- 4.1 Motivation . . . 32
- 4.2 MPPA/NoC architectures for the real-time . . . 35
  - 4.2.1 Structure of an MPPA . . . 35
  - 4.2.2 Support for real-time implementation . . . 36
  - 4.2.3 MPPA platform for off-line scheduling . . . 39
- 4.3 Software organization . . . 43
- 4.4 WCET analysis of parallel code . . . 44
- 4.5 Mapping (1) - MPPA-specific aspects . . . 45
  - 4.5.1 Resource modeling . . . 45
  - 4.5.2 Application specification . . . 46
  - 4.5.3 Non-functional properties . . . 48
  - 4.5.4 Scheduling and code generation . . . 49
- 4.6 Mapping (2) - Architecture-independent optimizations . . . 54
  - 4.6.1 Motivation . . . 54

4.6.2 Related work and originality 56
4.7 Conclusion 59

5 Automatic implementation of systems with complex functional and non-functional properties 61
5.1 Related work 62
5.2 Time-triggered systems 64
5.2.1 General definition 64
5.2.2 Model restriction 65
5.2.3 Temporal partitioning 66
5.3 A typical case study. 67
5.3.1 Functional specification 68
5.4 Non-functional properties 68
5.4.1 Period, release dates, and deadlines 68
5.4.2 Modeling of the case study 69
5.4.3 Architecture-dependent constraints 70
5.4.4 Worst-case durations, allocations, preemptability 71
5.4.5 Partitioning 72
5.5 Scheduling and code generation 72
5.6 Conclusion 73

Chapter 1

Introduction

Writing an habilitation thesis always involves a self-assessment of past research. In my case, this retrospect across the years revealed deep roots that may explain why, regardless of changes in research positions, *all* research I did fits in a well-defined research program that I would also like to continue in the future, a program I named *"Real-Time Systems Compilation"*.

This particularity of my work resulted in a particular structure for this thesis. To put in evidence the continuity between motivation, past research work, and research project, I concisely described them in a single chapter (this one). The remaining four chapters provide a more technical description of my most important results obtained after the PhD. Of these four chapters, the first is dedicated to an introduction to synchronous languages and formalisms, which are the semantic basis of my research work, written from the perspective of their use in the design of real-time embedded systems.

1.1 When it all began...

Like others of my generation, I discovered the world of computing on a *home computer*, a Sinclair ZX Spectrum clone. These were affordable, simple micro-computers[1] but already complex enough to exhibit the two sides of computing that have followed me to this day.

The rational, Apollonian side, the one I immediately liked, was that of programming. That of algorithms that I could write on a (paper) notebook using a clear syntax, before giving them to a computer. Of programs that during execution would either do exactly what I thought they will, or (if not) would allow me to find *my* error, usually a misunderstanding of some semantic rule. Using reason and the simple statements of BASIC I could at first create nice drawings, some music notes, then simple games. Later came more powerful computers, increasingly sophisticated languages and programming tools, and increasingly complex applications. But the basics, which I liked, remained largely the same.

[1] 8-bit Z80 CPU, 64kbytes RAM.

But there was also a dark, Dionysian side to home computers. That of computer games, programs of a special kind that I could not fully understand and control. This was a world of magic spells[2] transmitted from gamer to gamer and allowing one to obtain more (or infinite) lives, to change screen features, *etc.* Of gurus able to write (in assembly!) a new loader for some game in order to permanently alter its behavior, or add features to it.

Even though I didn't know it at the time, playing with home computer games was my first contact with the world of embedded computing. This side of computing fascinated me, and yet it made me uncomfortable. Attempts to understand why its various manipulations worked seemed doomed to failure, as they required detailed understanding of:

- The physical world with which the programs interacted. For instance, understanding data loading from audio cassettes required at least basic knowledge of sampling theory.
- Techniques for the efficient and real-time implementation of programs. This included detailed knowledge of the hardware, such as the functioning and real-time characteristics of video memory. It also included mastering the software development process, including low-level aspects such as assembly coding.

Searching, guessing, and trying seemed more important here than reasoning. And yet, over the years, I had to cope with this dark side again and again. First, on toy projects. For instance, creating a virus-like resident program on a PC required reprogramming the keyboard interrupt of my DOS/x86 system, while programming a two-wheeled Lego Mindstorms robot required me to understand the basics of the inverted pendulum, PID controllers and the functioning of sensor and actuator hardware. Later, through my research, I learned that industrial embedded systems designers shared the same problems, scaled up in complexity according to system size, embedding constraints (safety, low consumption, *etc.*), and industrial process considerations.

From my research I have also learned that the description of physical processes did not belong (any more) to the dark side. Well-defined languages, such as Simulink/Stateflow or LabView, had been introduced to allow their non-ambiguous description and analysis.

The implementation side also gained more detail. I was progressively able to grasp the complexity of the infrastructure that allowed sequential programs, written in BASIC, C, or Ada, to run correctly and efficiently. This infrastructure includes development tools (compiler, debugger) and system software (drivers, operating system, middleware). It also includes the standards on which these tools are based: programming languages, instruction set architectures (ISAs) such as x86 or ARMv5, application binary interfaces (ABIs) such as the ARM EABI, executable formats such as ELF, or even system-level standards such as POSIX or ARINC 653.

[2]Specific calls to the PEEK and POKE instructions that directly read and wrote specific memory addresses.

But even with this deeper understanding, the embedded design flow falls short of the expectations created by high-level sequential programming. Significant manual phases remain, where ensuring correctness and efficiency relies on the use of expert intervention (the modern equivalent of magic) to either manually transform the code or at least to validate it. In this context, it is only natural to ask the simple question that has guided my research: *what part of the embedded implementation process can be fully automated, in a way that ensures both correctness and efficiency?*

1.2 Overview of previous work

The question of automation in the embedded design process is general enough that I was able to follow it for my entire research career. Another aspect of my research work that never changed since the beginning of my PhD was the use of a specific tool: the synchronous, multi-clock, and time-triggered languages, presented in Chapter 2, which facilitate the formal specification and analysis of deterministic concurrent systems. Using these formalisms, I have considered three increasingly complex implementation problems that must be solved as part of the embedded design process:

Compilation of synchronous programs into efficient sequential (task) code. I started this line of work during my PhD, supervised by G. Berry and R. de Simone. I defined a technique for the compilation of imperative Esterel programs into fast and small sequential code. By introducing a series of optimizations based on static analysis of Esterel programs (using their rich structural information), I was able to produce code that still remains the fastest in terms of speed and a close contender in terms of size. To have a clear semantic definition of the data handling instructions of Esterel, and therefore be able to define the correctness of my compiler, I have also introduced a new operational semantics for the language. Main results on these topics are presented in my book "Compiling Esterel", co-written with S. Edwards and G. Berry [132]. The prototype compiler I wrote was transferred to industry (the Esterel Technologies company).

Automatic synthesis of optimal synchronization protocols for the concurrent implementation of synchronous programs. Large synchronous specifications are often implemented as a set of components (e.g. tasks, threads) running in an asynchronous environment. This can be done for execution or simulation purposes in a multi-thread, multi-task, or distributed context. To preserve the semantics of the initial synchronous specification, supplementary inter-component synchronizations may be needed, and for efficiency purposes it is important to keep synchronization at a minimum. As a post-doc, I started working on this problem with A. Benveniste and B. Caillaud, which had already defined endochrony. Endochrony is a property of synchronous components. When executed in an asynchronous environment, an endochronous component

remains deterministic without the need of supplementary synchronization, because sufficient synchronization is provided by the message exchanges already prescribed by the initial synchronous specification.

With my collaborators, I first determined that endochrony is a rather restrictive and non-compositional sufficient property. Then, we introduced the theory of weak endochrony, which characterizes exactly the components that need no supplementary synchronization [128, 126, 120]. Based on this theory, I defined algorithms for determining whether a synchronous program is weakly endochronous, and then a method for adding minimal (optimal) synchronization ensuring weak endochrony to an existing program [134, 118]. These algorithms use an original, compact representation of synchronization patterns of a synchronous program, which I defined. These algorithms were implemented in a prototype tool connected to the Signal/Polychrony toolset (post-doc of V. Papailiopoulou, collaboration with INRIA Espresso team) [118].

These results are presented in more detail in Chapter 3.

Efficient compilation of systems with complex functional and non-functional properties. Working with industrial partners from the embedded design field made me realize that my previous results addressed only particular, albeit important, aspects of a complex system (the synthesis of sequential tasks and the synthesis of communications). After joining the INRIA Aoste team as a permanent researcher I started investigating the system-level synthesis of real-time embedded systems,[3] and in particular that of systems relying on static (off-line) or time-triggered real-time scheduling. Such systems are used in safety-critical systems (avionics, automotive, rail) and in signal processing. I developed the conviction that building safe and efficient systems requires addressing two fundamental problems, which are only partially solved today:

- The seamless formal integration of full implementation flows going all the way from high-level specification (e.g. Scade, Simulink) to running implementation (code executing on the platform *and* platform configuration).

- Fast and efficient synthesis with full error traceability, which allows the use of a trial-and-error design style aimed at maximizing engineer productivity.

To solve these two problems, I have designed and built, with my students and post-docs, the LoPhT real-time systems compiler [38, 37, 73, 36, 124, 130]. By using fast allocation and scheduling heuristics to ensure scalability, LoPhT takes inspiration from previous work in the fields of off-line real-time scheduling, optimized compilation, and synchronous language analysis and implementation. But LoPhT goes beyond previous work by carefully integrating semantically and algorithmically aspects that were previously considered separately, such as

[3] A subject well-studied in the team by both Y. Sorel, author of the AAA methodology [76], and R. De Simone, whose research interest in modeling time was materialized, among others, in a significant contribution to the time model of the UML MARTE standard [53].

the fine semantic properties of the high-level specifications [130], detailed descriptions of the execution platforms (hardware, OS/libraries, execution model) [124, 36, 38], and complex non-functional specifications covering all the modeling needs of realistic systems [38]. For instance, LoPhT combines in a single tool the use of a classical real-time scheduling algorithm (deadline-driven scheduling) with classical compiler optimizations (*e.g.* software pipelining [37, 38]), domain-specific optimizations (safe double reservation based on predicate analysis [130]), and platform-specific optimizations (minimizing the number of tasks and partition switches for ARINC 653 systems [38], pre-emptive communications scheduling for many-core and TTEthernet-based networks [124, 36], *etc.*). Combined with precise time accounting, the integration of these optimizations allows the generation of efficient code while providing formal correctness guarantees.

I have dedicated special attention to ensuring that the platform models used by the scheduling algorithms are conservative abstractions of the actual platforms. To do this, I have initiated collaborations that allowed us to explore the design of execution platforms with support for off-line real-time scheduling. Such platforms allow the construction of applications that are both highly efficient and temporally predictable [55, 35, 124]. Together with my collaborators, I have determined that such architectures enable precise worst-case execution time analysis for parallel software [133] and efficient application mapping [36]. I have also initiated industrial collaborations meant to ensure that LoPhT responds to industry needs, and to promote its use [73, 38, 49].

These results are presented in more detail in Chapters 4 and 5. The first one considers a more compilation-like point of view by focusing on fine-grain architecture detail and by considering mapping problems where the objective is to optimize simple metrics. While providing hard real-time guarantees, the methods presented in this chapter do not consider real-time requirements. Subjects covered in this chapter are the mapping of applications to many-cores, the use of advanced compiler optimizations in off-line real-time scheduling, and the worst-case execution time analysis of parallel code.

Non-functional requirements of multiple types (real-time, partitioning, preemptability) are considered in Chapter 5, in conjunction with time-triggered execution targets. This completes the definition of our real-time systems compilation approach.

1.3 Research project: Real-time systems compilation

The implementation of complex embedded software relies on two fundamental and complementary engineering disciplines: *real-time scheduling* and *compilation*. Real-time scheduling covers[4] the upper abstraction levels of the implementation process, which determine how the *functional specification* is transformed

[4]Together with other disciplines such as systems engineering, software engineering, *etc.*

into a set of *tasks* and then determine how the tasks must be *allocated* and *scheduled* onto the *resources* of the execution platform in a way that ensures *functional correctness* and the respect of *non-functional requirements*. By comparison, compilation covers the low-level code generation process, where each task (a piece of sequential code written in C, Ada, *etc.*) is transformed into machine code, allowing actual execution.

In the early days of embedded systems design, both high-level and low-level implementation activities were largely manual. However, this is no longer the case in the low level, where manual assembly coding has been almost completely replaced by the combined use of programming languages such as C or Ada and compilers [60]. This shift towards high-level languages and compilation allowed a significant *productivity gain* by ensuring that source code is *safer* and *more portable*. As compiler technology improved and systems became more complex in both hardware and software, compilation has also approached the *efficiency* of manual assembly coding, and in most cases outperformed it.

The widespread adoption of compilation was only possible due to the early adoption of standard interfaces that allowed the definition of economically-viable compilation tools with a large-enough user base. These interfaces include not only the programming languages (C, Ada, *etc.*), but also relatively stable microprocessor instruction set architectures (ISAs) or executable code formats like ELF.

The paradigm shift towards fully automated code generation is far from being completed at the system level. Aspects such as the division of the functional specification into tasks, the allocation of tasks to resources, or the configuration of the real-time scheduler are still performed manually for most industrial applications. Furthermore, research in real-time scheduling has largely followed this trend, with most (but not all) effort still invested into verification-based approaches aimed at proving the *schedulability* of a given system (and into the definition of run-time mechanisms improving resource use).

This slow adoption of automatic code generation can be traced back to the slower introduction of standard interfaces allowing the definition of economically-viable compilers. This also explains why real-time scheduling has historically dedicated much of its research effort to verifying the correctness of very abstract and relatively standard implementation models (the task models). The actual construction of the implementations and the abstraction of these implementations as task models drew comparatively less interest, because they were application-dependent and non-portable.

But if standardization and automation advanced slower, they advanced nevertheless. Functional specification languages such as Simulink, LabVIEW, or SCADE have been introduced in the mid-1980s, which allowed the gradual definition of techniques for the synthesis of functionally-correct sequential or even multi-task embedded code (but without real-time guarantees). The next major step came in the mid-1990s, when execution platforms have been standardized in fields such as avionics (IMA/ARINC 653) and automotive (OSEK/VDO, then AUTOSAR). This second wave of standardization already allowed the industrial introduction of automatic tools for the (separate) synthesis of processor

schedules or network schedules.

The research community went even farther and proposed real-time implementation flows that automatically produced running real-time applications [76, 37, 36, 50] where the processor and network schedules are jointly computed using a *global optimization approach* that results in better resource use. Of course, more work is needed to ensure the industrial applicability of such results. For instance, the aforementioned techniques could not handle all the complexity of IMA avionics systems, which involve functional specifications with multiple execution modes, multi-processor architectures with complex interconnect networks, and complex non-functional requirements including real-time, partitioning, preemptability, allocation, *etc.*

This explains why, to this day, the design and implementation of industrial real-time systems remains to a large extent a craft, with significant manual phases. But a revolution is brewing, driven by two factors:

- Automation can no longer be avoided, as the complexity of systems steadily increases in both specification size (number of tasks, processors, etc.) and complexity of the objects involved (dependent tasks, multiple modes and criticalities, novel processing elements and communication media...).
- Fully automated implementation is attainable for industrially significant classes of systems, due to significant advances in the standardization of both specification languages and of implementation platforms.

To allow the automatic implementation of complex embedded systems, I advocate for a *real-time systems compilation* approach that combines aspects of both real-time scheduling and (classical) compilation. Like a classical compiler such as GCC, a real-time systems compiler should use fast and efficient scheduling and code generation heuristics, to ensure scalability.[5] Similarly, it should provide traceability support under the form of informative error messages enabling an incremental trial-and-error design style, much like that of classical application software. This is more difficult than in a classical compiler, given the complexity of the transformation flow (creation of tasks, allocation, scheduling, synthesis of communication and synchronization code, *etc.*), and requires a full formal integration along the whole flow, including the crucial issue of correct hardware abstraction.

A real-time systems compiler should perform precise, conservative timing accounting along the whole scheduling and code generation flow, allowing it to produce safe and tight real-time guarantees. More generally, and unlike in classical compilers, the allocation and scheduling algorithms must take into account a variety of non-functional requirements, such as real-time constraints, criticality/partitioning, preemptability, allocation constraints, *etc.* As the accent is put on the respect of requirements (as opposed to optimization of a metric, like in classical compilation), resulting scheduling problems are quite different.

Together with my students, I have defined and built such a real-time systems compiler, called LoPhT, for statically scheduled real-time systems. While first

[5] Exact application mapping techniques do not scale [72].

results are already here [37, 38, 73, 36, 35], I believe that the work on real-time systems compilation is only at its beginning. It must be extended to cover more execution platforms, and we are currently working on porting LoPhT on the Kalray MPPA256 many-core and on TTEthernet-based time-triggered systems.

Efficiency is also a critical issue in practical systems design, and we must invest even more in the use of classical optimizations such as loop unrolling and inline expansion. as well as new optimizations specific to the real-time context and to each platform in particular. To cover these needs, we must also go beyond fully static/offline scheduling, but while remaining fully formal, automated, and safe.

Ensuring the safety and efficiency of the generated code cannot be done by a single team. I am actively promoting the concept of real-time systems compilation in the community, and collaborations on the subject will have to cover at least the following subjects: the interaction between real-time scheduling and WCET analysis, the design of predictable hardware and software architectures, programming language support for efficient compilation, and formally proving the correctness of the compiler. Of course, the final objective is that of promoting real-time systems compilation in the industry, and to this end I actively seek industrial case studies and disseminate our work towards industry.

From a methodological point of view, my research will continue on its current trend of combining concepts and methods from 3 different communities: compilation, real-time scheduling, and synchronous languages. I am fully aware of the long-standing separation between the fields of compilation and real-time scheduling.[6] However, I believe that the original reasons of this separation[7] are less and less true today.[8] The convergence between these two communities seems to me inevitable in the long run, and my work can be seen as part of the much-needed mutualization of resources (concepts and techniques) between the two fields. My work also shows that synchronous languages should play an important role in the convergence between real-time scheduling and compilation. First of all, as a common ground for formal modeling. Indeed, synchronous formalisms are natural extensions of formalisms of both real-time scheduling (the dependent task graphs) and compilation (static single assignment representations and the data dependency graphs). But beyond being a mere common formal ground, previous work on synchronous languages also provides powerful techniques for the modeling and analysis of complex control structures that are used in embedded systems design.[9]

[6] Publishing has been quite a struggle for this reason.

[7] Focus on sequential code and static scheduling in the compilation community, focus on dynamic, multi-task code in real-time scheduling.

[8] Compilation, for instance, considers dynamically-scheduled targets such as GPGPUs, and some algorithms perform precise timing accounting, like in software pipelining. At the same time, real-time scheduling is considering with renewed interest statically-scheduled targets (due to industrial demand).

[9] By means of so-called clocks and delays, presented in the next chapter.

Chapter 2

Introduction to synchronous languages

As evidenced by the publication record, all three of the originary synchronous languages (Esterel, Lustre, and Signal) are the product of the 1980s real-time community [21, 20, 99], where they were introduced to facilitate the high-level specification of complex *real-time embedded control systems*.

An embedded control system aims to control, in the sense of *automatic control theory*, a *physical process* in order to lead it towards a given state. The physical process and the control system, which usually form a closed loop [56], are in the beginning specified in continuous time in order to be analyzed and simulated. Then, the control system is *discretized* in order to allow its implementation on the embedded *execution platform*. Fig. 2.1 describes the interactions between the discrete control system and the physical process. The embedded control system obtains its discrete-time inputs through sensors equipped with analog-digital converters (ADC). The discrete outputs of the embedded system are transformed by the digital-analog converters (DAC) of the actuators into continous-time feedback to the physical process. Both inputs and outputs can be implemented using event-driven or periodic sampling (time-triggered) mechanisms.

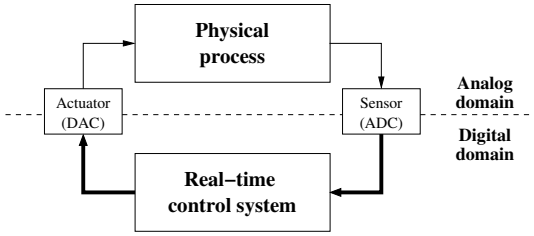

Figure 2.1: Closed loop control system

Synchronous languages were introduced in order to specify discretized control systems, which are reactive, real-time systems. In a reactive system [82, 78], execution can be described as a set of interactions between the system and the physical process. Each acquisition of data by the sensors is followed by a reaction of the control system, which consists in performing some computations and then updating the actuators. Multiple reactions may be executed at the same time, competing for the same execution resources, a property known as concurrency.

Real-time systems are reactive systems where reactions are subject to timing constraints. These constraints are determined by the control engineers during discretization of the control system. The constraints may concern the sampling periods of the sensors and actuators and/or the latencies (end-to-end delays) of the reactions. The sampling constraints on the sensors and actuators determine the periods of the computation functions (tasks) that depend on or drive sensing and actuation. The latency constraints are applied to chains of computation functions, which may have a sensor or an actuator as extremity. A deadline is a particular case of latency constraint that is applied to a single computation function, for instance the code controlling a sensor or a digital filter.

Reactive and real-time control systems have particular specification needs. To describe the reactive aspects, synchronous languages offer syntactical constructs allowing the specification of order (dependency, sequence), concurrency (parallelism), conditional execution and simultaneity relations between operations of the system (data acquisitions, computation functions, data transfers and actuator updates).

For the non-functional specification of the real-time aspects, the synchronous languages implicitly define or allow the explicit definition of one or more discrete time bases, called clocks. A clock describes a finite or infinite sequence of events in the execution of the system. Thus, each clock divides the execution of the system into a series of execution steps, which are sometimes called logical instants, reactions, or computation instants. We can associate a clock with each periodic event (*e.g.* periodic timer), sporadic event (*e.g.* the top dead center, or TDC, of a piston in a combustion engine), aperiodic event (*e.g.* button press), or simply with an internal event of the control system, which is built from other internal or external events and therefore depends on other clocks.

Clocks are so-called *logical* time bases. This means that the synchronous languages allow the specification of order relations between events associated with these clocks, but do not offer support for the analysis of the relations between physical quantities they may represent (except through specific extensions detailed below). Clocks associated with physical quantities are called physical clocks. For instance, an engine control system may have a physical clock associated to a timer and another physical clock associated with the the TDC event. By taking into account the maximum speed of the engine, we can determine the maximal duration (in time) between two events of the TDC clock, thus relating events of the two clocks. Such an analysis requires the application of physical theories, in addition to the theory of synchronous languages.

The execution of every operation of a synchronous system has to be synchro-

nized with respect to at least one clock. Real-time information coming from the control specification (periods, deadlines, latencies) cannot be directly associated to operations. Instead, it is associated to the clocks, which in turn drive the execution of operations.

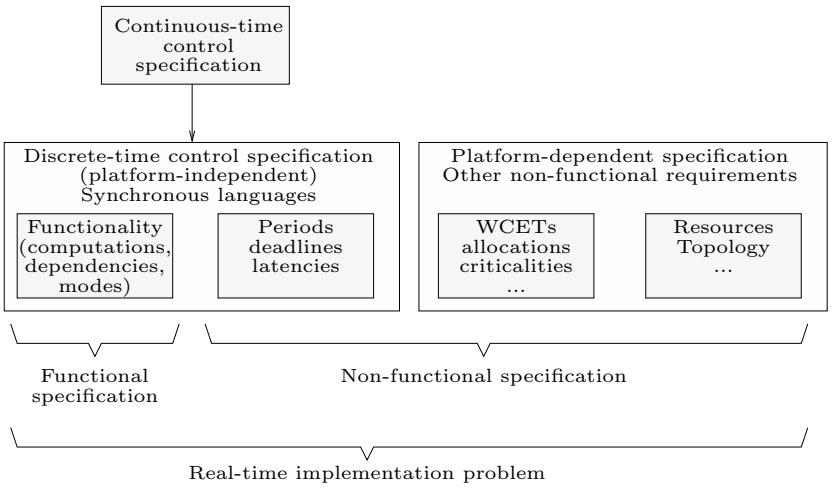

Figure 2.2: Scope of application of synchronous languages in real-time systems specification

As shown in Fig. 2.2, the specification of a real-time implementation problem does not only include the platform-independent discrete-time controller specification, provided under the form of a synchronous program. It also includes non-functional requirements coming from other engineering disciplines (such as criticalities) and the constraints related to the implementation of the control system on an embedded execution platform. These platform-dependent constraints include the definition of the resources of the platform, the worst-case execution time estimations (WCETs) of computations on the CPUs, the worst-case durations of communications over the buses (WCCT), the allocation constraints, *etc.*

Criticalities and platform-dependent information are not part of the discrete-time controller specification, and synchronous languages are not meant to represent them. In other terms, synchronous languages alone are not equiped to allow the specification and analysis of all the aspects of a real-time embedded implementation problem. Some synchronous languages allow the specification of platform-related properties through dedicated extensions that will be discussed later in this chapter.

Ignoring platform-related aspects is one of the key points of synchronous languages. Ignoring execution durations means that we may assume the computation of each reaction to take 0 time, so that its inputs and outputs are simultaneous (synchronous) in the discrete time scale (clock) that governs the

execution of the reaction. This *synchrony hypothesis*, which gives its name to the synchronous model, is naturally inherited through discretization from continuous-time modeling, where it is implicitly used.

Implementing a specification relying on the synchrony hypothesis amounts to solving a scheduling problem which ensures that:

- The resources of the execution platform allow each reaction to terminate before their outputs are needed – either as input to other reactions or to drive the actuators.

- All period and latency requirements specified by the control engineers are satisfied.

As part of the synchrony hypothesis, we also require that a reaction has a bounded number of operations. This assumption ensures that, independently of the execution platform, the computation of a reaction is always completed in bounded time, which allows the application of real-time schedulability analysis.

Under the synchrony hypothesis, all computations of a reaction are synchronous, in the sense that they are not ordered in the discrete time scale defined by the clock of the reaction. However, their execution has to be causal:

- Two reads of the same variable/signal performed during the same reaction must always provide the same result.

- If a variable/signal is written during a reaction, then all reads inside the reaction will produce the same value. No variable/signal should be written twice during a reaction, or otherwise it must be specified which of the writes gives the variable/signal its value. This amounts to requiring that reading a variable/signal is performed in a reaction only after all write operations on the variable/signal have been completed.

Causality ensures functional determinism in the presence of concurrency. It ensures that executing the computations and communications of a reaction will always produce the same result, for any scheduling of the operations that satisfies the data and control dependencies. In a causal system a set of inputs will always produce the same set of outputs. This property is important in practice, since it simplifies the costly activities of verification and validation (test, formal verification, *etc.*), as well as debugging.

These four ingredients – clocks, synchrony hypothesis, causality, and functional determinism – define a formal base that is common to all synchronous languages. It ensures strong semantic soundness by allowing universally recognized mathematical models such as the Mealy machines and the digital circuits to be used as supporting foundations. In turn, these models give access to a large corpus of efficient optimization, compilation, and formal verification techniques. The synchronous hypothesis also guarantees full equivalence between various levels of representation, thereby avoiding altogether the pitfalls of non-synthesizability of other similar formalisms.

2.1 Synchronous languages

Structured languages have been introduced for the modeling and programming of synchronous applications. From a syntactical point of view, each one of them provides a certain number of constructions facilitating the description of complex systems: concurrency, conditional execution and/or modes of execution, dependencies and clocks allowing to describe complex temporal relations such as multiple periods. This allows the incremental (hierarchical) specification of complex behaviors from elementary behaviors (computation functions without side effects and with bounded durations). The concision and the deterministic semantics of synchronous specifications make them a good starting point for design methodologies for safe systems, where a significant part of the time budget is devoted to formal analysis and testing.

Language	Imperative/ Data Flow	Base clock(s)	"Physical" time	Real-time analysis
Esterel/SSM	I(+DF)	Single	–	
Lustre/Scade	DF(+I)	Single	–	
TAXYS	I	Single	APODW	WCRT, sched
Lucy-n	DF	Affine	–	
SynDEx	DF	Affine	PW (D=P)	WCRT, sched
Giotto	DF	Affine	P (D=P)	
Prelude	DF	Affine	PODW	WCRT, sched
Signal	DF(+I)	Multiple	–	
EmbeddedCode	I	Multiple	AD	
ΨC	I+DF	Multiple	APODW	WCRT, sched
SciCos	DF	Multiple	C	
Zélus	DF	Multiple	C	

Table 2.1: Classification of synchronous languages. I = Imperative, DF = Data flow, P = periodic activations, A = aperiodic activations, D = Deadlines, O = Offsets, W = durations, C = continuous time.

However, beyond these aspects, each of the synchronous languages has original points and particular uses, and therefore a classification is required. Table 2.1 summarizes this classification along 4 criteria.

Programming paradigm. According to the programming paradigm, synchronous languages are divided into two large classes: declarative data-flow languages and imperative languages. Declarative languages, which include data-flow languages, focus on the definition of the function to be computed. Imperative languages describe the organization of the operations needed to compute this function (computations, decisions, inputs/outputs, state changes). Among the synchronous languages, Esterel[21], SyncCharts[7], ΨC[42, 43], and the embedded code formalism [86] are classified as imperative, while the Lustre/SCADE[20], Signal/Polychrony[99, 19], SynDEx[76, 94], Giotto[85], and Prelude[116] are

classified as data-flow. The SciCos[32] and Zélus[28] languages are data-flow languages, with the particularity of being hybrid languages, which allow the representation of both continuous-time and discrete-time control systems.

Data-flow languages are syntactically closer to synchronous digital circuits and to real-time dependent task models. In these languages, the concurrency between operations is only constrained by explicit data dependencies. Data-flow languages are used to highlight the flow of information between parts of an application, or its structuring into tasks.

Imperative languages are syntactically closer to (hierarchical) synchronous automata. They are generally used to represent complex control structures, such as those of an operating system scheduler. Besides concurrency, they allow the specification of operation sequencing and offer hierarchical constructs allowing to stop or resume a behavior in response to an internal or external signal. The data dependencies are often represented implicitly, using shared variables, instead of explicit data dependencies.

The first synchronous languages could be easily classified as imperative (Esterel) or data-flow (Lustre, Signal). However, the successive evolutions of these languages have made classification more difficult. For instance, the data-flow language Scade/Lustre has incorporated imperative elements, such as the state machines, whereas an imperative language such as Esterel has incorporated declarative elements such as the sustained emit instruction. This is why, in our table, some languages belong to both classes.

Number of time bases. A second criterion of classification of synchronous languages is related to the number of time bases that can be defined and used in a program. In Lustre/SCADE and Esterel, which we call single-clock, a single logical time base exists, called global clock or base clock. All other time bases (clocks) are explicitly derived from the base clock by sub-sampling.

The Signal and ΨC languages do not have this limitation. They allow the definition of several logical time bases. As explained above, an automotive engine control application may have two base clocks, one corresponding to time and the other to the rotation of the engine (TDC), and these two clocks cannot be defined from one another. Having two base clocks allows the operations to be ordered with respect to events of two different discrete time bases, which may facilitate both modeling an analysis.[1] The languages allowing the definition of multiple, independent clocks are called polychronous or multi-clock.

Between single-clock languages and multi-clock languages, we identify an intermediate class of languages that allow the definition of several time bases, but require that as soon as two clocks are used by a same operation, they become linked by a relation allowing to completely order their logical instants in a unique way. Therefore, a global clock can be built unambiguously for every program from the time bases specified by the programmer.[2] However, it is

[1] Buiding a single-clock model of an application is always possible[79], but analysis may be more complicated.

[2] More precisely, we can build such a clock if the program cannot be divided into completely

often more interesting to not build this global clock and instead apply specific analysis directly on the clocks specified by the programmer. For instance, the languages Giotto, Lucy-n, Prelude and SynDEx allow the definition of clocks linked in period and phase (offset) by affine relations. These languages allow a more direct description of periodic real-time task systems with different periods [116, 51, 94].

Modeling of "physical" time. The third classification criterion we use is the presence in the language of extensions allowing the description of physical time. This concept appears naturally in languages allowing the specification of continuous-time systems, like in SciCos or Zélus [32, 28]. However, we are more interested here by languages aiming directly at the specification of a real-time implementation problem. This requires concepts such as periodic and aperiodic activations, deadlines, offsets and execution times. These extensions allow the application of various real-time analyses: worst-case response time analysis, schedulability analysis, or even the synthesis of schedules or the adjustment of parameters of the scheduler of a real-time operating system.

Enforcement of synchrony hypothesis. Our final classification criterion concerns the enforcement of the synchrony hypothesis. The Esterel, Lustre, Signal, and SynDEx languages require strict adherence to it. The computation and data transfer operations are semantically infinitely fast, so that a reaction must always terminate before the beginning of the next one. The execution of the system can therefore be seen as the totally ordered sequence of reactions.[3] In particular, every operation (computation or communication), independently of its clock, can be and must be terminated in the clock cycle where it began, before the beginning of the next clock cycle. If we associate real-time periods to logical clocks, this assumption implies that an operation (computation or communication) cannot have a duration longer than the greatest common divisor (GCD) of the periods of the system.

However, the description of real-time systems often implies so-called long tasks with a duration longer than the GCD of the periods. Representing such tasks in a synchronous formalism requires constraining the synchronous composition to ensure that an operation takes more than one logical instant. One way to do it is by systematically introducing explicit delays between the end of an operation and the operations depending on it. These delays explicitly represent the time (in clock cycles) reserved for the execution of the operation. Introducing such delays manually may be tedious, and some languages, such as Giotto, Prelude, and SynDEx have proposed dedicated constructs with the same effect. In Giotto, the convention is that the outputs of a task remain available during the clock cycle following the one where the operation started, in the time base given by the clock associated with the task. Prelude is more expressive. It allows the definition of delays shorter than one clock cycle by refining the clock

independent parts.

[3] This is true even for multi-clock languages.

of the operation and then working in this refined time base. SynDEx proposes an intermediate solution.

2.2 Related formalisms

Of course, synchronous languages are only one of the classes of formalisms used in embedded control system design. For instance, in traditional real-time systems design, two levels of representation are particularly important: real-time task models [102, 14], which serve to perform the real-time scheduling analysis (feasibility or schedulability), and the low-level implementation code, provided in languages such as C or assembly.

Real-time task models are not designed as full-fledged programming languages, focusing only on the definition of properties that will be exploited by classical schedulability analysis techniques. Among these properties: the organization of computations into tasks, the periods, durations, and deadlines of tasks, and sometimes their dependencies or exclusion relations. By comparison, synchronous languages are full-fledged programming languages that can serve both as support for real-time scheduling analyses and as task-level and system-level programming languages. They allow, for instance, the full specification of a task functionality, followed by fully automatic generation of the low-level task code. They also allow the specification of full systems including tasks, OS, and hardware for simulation, formal analysis, or to allow the synthesis of task sequencing and synchronization code. Thus, while remaining at a high abstraction level and focusing on the specification of platform-independent functional and timing aspects, synchronous languages allow the automatic synthesis of an increasing part of the low-level code, especially in critical embedded control systems.

Like synchronous languages, the synchronous data-flow (SDF) [LEE 87] and derived formalisms (such as CSDF [22], SigmaC[74], or StreamIt[5]) feature a cyclic execution model. The difference is that repetitions (cycles) of the various computations and communications are not synchronized along global time references (clocks). Instead, the execution of each data-flow node is driven by the arrival of input data along lossless FIFO channels (a form of local synchronization).

The pair of formalisms Simulink/StateFlow [27, 46] is the *de facto* standard for the modeling of control systems. These languages share with synchronous languages a great deal of their basic constructs: the use of logical and physical time bases, the synchrony hypothesis, a definition of causality and even a good part of the language constructs. However, the differences are also great: synchronous languages aim to give unique and deterministic semantics to every correct specification, and thus they aim to ensure the equivalence between analysis, simulation and execution of the implemented system. The objective of Simulink (as its name indicates) is to allow the simulation of control systems, whether they are specified in discrete and/or continuous time. The definition of causal dependencies is clear, but it depends on the chosen simulation mode,

and the number of simulation options is such that it is sometimes difficult to determine which rules apply. To accelerate the simulations, there are options that explicitly allow for non-determinism. Finally, the determinism of the simulation is sometimes only acquired through the use of rules depending on the relative position of the graphical objects of a specification (in addition to the classical causality rules). By comparison, the semantics of a synchronous program only depends on the data dependencies between operations, which allows to preserve more concurrency and therefore give more freedom to the scheduling algorithms.

The definition of a synchronized cyclic execution on physical or logical time bases is also shared by formalisms such as StateCharts [81] or VHDL/VERILOG [90]. Like synchronous languages, these formalisms define a concept of (logical) execution time and allow a complex propagation of control within these instants. However, synchronous causality (and thus determinism) is not always required.

Chapter 3

Automatic synthesis of optimal synchronization protocols

Synchronous programming is nowadays a widely accepted paradigm for the design of critical applications such as digital circuits or embedded real-time software [18, 122], especially when a semantic reference is sought to ensure the coherence between the implementation and the various analyses and simulations.

But building concurrent (distributed, multi-task, multi-thread) implementations of synchronous specifications remains an open and difficult subject, due to the need of preserving the global synchronizations specific to the model. Synchronization artifacts need most of the time be preserved, at least in part, in order to ensure functional correctness when the behavior of the whole system depends on properties such as the arrival order of events on the various communication media or the passage of time, as measured by the presence or absence of events in successive reactions.

Ensuring synchronization in concurrent implementations can be done in two fundamentally different manners:

- **Delay-insensitive**[1] synchronization protocols make no hypothesis on the *real-time duration* of the various computations or communications. Under this hypothesis, detecting the sending order of two events arriving of different communication media is *a priori* impossible, as is determining that a signal is absent in a reaction.[2] Delay-insensitive synchronization protocols can only rely on the ordering of events imposed by the various

[1] In the sense of delay-insensitive algorithms [15], sometimes also called self-timed, or scheduling-independent.

[2] Because each computation or communication can take an arbitrary, unbounded time. Another consequence of this property is the impossibility of consensus in faulty asynchronous systems [63].

system components, such as the sequencing of message transmissions on each bus, or the sequencing of computations on each processor.

- **Delay-sensitive** synchronization protocols allow the use of hypotheses on the real-time durations of the various computations and communications. In time-triggered systems [38], for instance, time-based synchronization is dominant and great care must be taken to ensure that a global time reference is available, with good-enough precision and accuracy.

Time-triggered delay-sensitive systems will be the focus of Chapter 5. In the current chapter I consider the problem of constructing delay-insensitive implementations of deterministic synchronous specifications. Using such delay-insensitive protocols in the construction of embedded real-time systems can be useful in two circumstances:

- When building non-real-time or soft real-time systems where the accent is put on computational efficiency, rather than on the respect of real-time requirements. In such systems, tasks are often executed on a platform whose temporal behavior cannot be precisely predicted due to reasons such as an unknown number of cores, the use of a dynamic fair scheduler, or the unknown cost of system software (drivers and OS).

- When building hard real-time systems where the platform provides predictability guarantees, it may be useful to enforce a separation of concerns between functional correctness and real-time correctness issues in the design flow. A delay-insensitive functionally-correct implementation may be first used for functional simulations, before being provided as input to the allocation and scheduling phases that configure the execution platform. Such an approach is taken in SynDEx [76] and OCREP [41].

In both cases, the use of delay-insensitive synchronization provides guarantees of functional correctness independently from timing correctness.

Possibly the most popular approach of building deterministic delay-insensitive concurrent systems is the one based on the Kahn principle, which provides the theoretical basis for building Kahn process networks (KPN) [91, 105, 123]. The Kahn principle states that interconnecting deterministic delay-insensitive components by means of deterministic delay-insensitive communication lines always results in a deterministic delay-insensitive system. This provides a solid two-stage methodology for building delay-insensitive systems. The first stage consists in building deterministic delay-insensitive components, which are then incrementally composed together in phase two.

The work I present in this chapter has focused on applying this two-stage approach to the implementation of synchronous specifications. The problem I considered is that of building synchronous components (programs) that can function as deterministic delay-insensitive systems when the global clock synchronization is removed. The main difficulty here is transforming general synchronous components into delay-insensitive ones by adding *minimal* synchronization to their interfaces.

3.1 Semantics of a simple example

I use a small, intuitive example to present the problem, the desired result, and the main implementation issues. The example, pictured in Fig. 3.1, is a reconfigurable filter (in this case a simple adder, but similar reasoning can be applied to other filters, such as the FFT). In this example, two independent single-word adders can be used either independently, or synchronized to form a double-word adder. The choice between synchronized and non-synchronized mode is done using the `SYNC` signal. The carry between the two adders is propagated through the Boolean wire `C` whenever `SYNC` is present. To simplify figures and notations, we group both integer inputs of `ADD1` under `I1`, and both integer inputs of `ADD2` under `I2`. This poses no problem because from the synchronization perspective of this chapter the two integer inputs of an adder have the same properties.

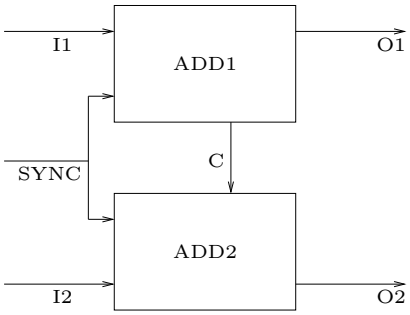

Figure 3.1: Data-flow of a configurable adder.

Time is discrete, and executions are sequences of *reactions*, indexed by a *global clock*. Given a synchronous program, a reaction is a valuation of its *input, output and internal (local) signals*. Fig. 3.2 gives a possible execution of our example. We shall denote with $\mathcal{V}(P)$ the finite set of signals of a program P. We shall distinguish inside $\mathcal{V}(P)$ the disjoint sub-sets of *input* and *output signals*, respectively denoted $\mathcal{I}(P)$ and $\mathcal{O}(P)$.

Reaction	1	2	3	4	5	6	7
I1	(1,2)	$*$	(9,9)	(9,9)	$*$	(2,5)	$*$
O1	3	$*$	8	8	$*$	7	$*$
SYNC	$*$	$*$	$\bullet$	$*$	$*$	$\bullet$	$*$
C	$*$	$*$	1	$*$	$*$	0	$*$
I2	$*$	$*$	(0,0)	(0,0)	$*$	(1,4)	(2,3)
O2	$*$	$*$	1	0	$*$	5	5

Figure 3.2: A synchronous run of the adder

If we denote with EXAMPLE our configurable adder, then

$$\begin{aligned}\mathcal{V}(\texttt{EXAMPLE}) &= \{\texttt{I1}, \texttt{I2}, \texttt{SYNC}, \texttt{O1}, \texttt{O2}, \texttt{C}\}\\ \mathcal{I}(\texttt{EXAMPLE}) &= \{\texttt{I1}, \texttt{I2}, \texttt{SYNC}\}\\ \mathcal{O}(\texttt{EXAMPLE}) &= \{\texttt{O1}, \texttt{O2}\}\end{aligned}$$

All signals are typed. We denote with $\mathcal{D}_S$ the domain (set of possible values) of a signal S. Not all signals need to have a value in a reaction, to model cases where only parts of the program compute. We will say that a signal is *present* in a reaction when it has a value in $\mathcal{D}_S$. Otherwise, we say that it is *absent*. Absence is simply represented with a special value $*$, which is appended to all domains $\mathcal{D}^*_{\mathtt{S}} = \mathcal{D}_{\mathtt{S}} \cup \{*\}$.

Formally, a reaction of a program P is a valuation of all the signals S of $\mathcal{V}(P)$ into their extended domains $\mathcal{D}^*_{\mathtt{S}}$. We denote with $\mathcal{R}(P)$ the set of all reactions of P. Given a set of signals $\mathcal{V}$, we denote with $\mathcal{R}(\mathcal{V})$ the set of all possible valuations of the signals in $\mathcal{V}$. Obviously, $\mathcal{R}(P) \subseteq \mathcal{R}(\mathcal{V}(P))$. In a reaction r of a program P, we distinguish the *input event*, which is the restriction $r\mid_{\mathcal{I}(P)}$ of r to input signals, and the *output event*, which is the restriction $r\mid_{\mathcal{O}(P)}$ to output signals.

In many cases we are only interested in the presence or absence of a signal, because it transmits no data, just synchronization (or because we are only interested in synchronization aspects). To represent such signals, the Signal language [77] uses a dedicated type named `event` of domain $\mathcal{D}_{\texttt{event}} = \{\bullet\}$. We follow the same convention: In our example, `SYNC` has type `event`. The types of the other signals in Fig. 3.2 are `SYNC:event`; `O1,O2:integer`; `I1,I2:integer_pair`; `C:boolean`.

To represent reactions, we use a *set-like convention* and omit signals with value $*$. Thus, reaction 4 is denoted $(\texttt{I1}^{(9,9)}, \texttt{O1}^{8}, \texttt{I2}^{(0,0)}, \texttt{O2}^{0})$.

3.2 Problem definition

We consider a synchronous program, and we want to execute it in an asynchronous environment where inputs arrive and outputs depart via asynchronous FIFO channels with uncontrolled (unbounded, but finite) communication latencies. To simplify, we assume that we have exactly one channel for each input and output signal of the program. We also assume a very simple correspondence between messages on channels and signal values: Each message on a channel corresponds to exactly one value (not absence) of a signal in a reaction. No message represents absence.

The execution machine driving the synchronous program in the asynchronous environment cyclically performs the following 3 steps:

1. assembling asynchronous input messages arriving onto the input channels into a synchronous input event acceptable by the program,

2. triggering a reaction of the program for the reconstructed input event, and

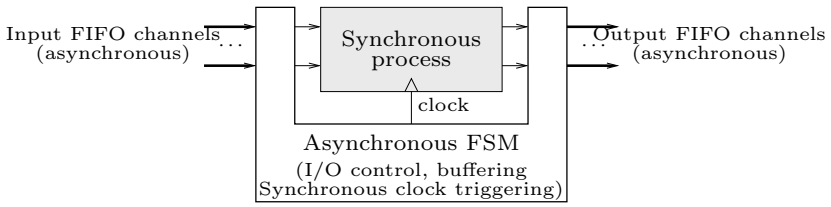

Figure 3.3: GALS wrapper driving the execution of a synchronous program in an asynchronous environment

3. transforming the output event of the reaction into messages onto the output asynchronous channels.

Fig. 3.3 provides the general form of such an execution machine, which is basically a wrapper transforming the synchronous program into a globally asynchronous, locally synchronous (GALS) [44] component that can be used in a larger GALS system. The actual form of the asynchronous finite state machine (AFSM) implementing the execution machine, and the form of the code implementing the synchronous program depends on a variety of factors, such as the desired implementation (software or hardware), the asynchronous signaling used by the input and output FIFOs, the properties of the synchronous program, *etc.*

In order to achieve deterministic execution,[3] the main difficulty lies in step (1) above, as it involves the potential reconstruction of signal absence, whereas absence is meaningless in the chosen asynchronous framework. Reconstructing reactions from asynchronous messages must be done in a way that ensures global determinism, regardless of the message arrival order. This is not always possible. Assume, like in Fig. 3.4, that we consider the inputs and outputs of Fig. 3.2 without synchronization information.

I1	(1,2)	(9,9)	(9,9)	(2,5)
O1	3	8	8	7
SYNC		•	•	
C		1	0	
I2	(0,0)	(0,0)	(1,4)	(2,3)
O2	1	0	5	5

Figure 3.4: Corresponding asynchronous run of our example. No synchronization exists between the various signals, so that correctly reconstructing synchronous inputs from the asynchronous ones is impossible

The adder `ADD1` will then receive the first value $(1, 2)$ on the input channel `I1` and • on `SYNC`. Depending on the arrival order, which cannot be determined, any

[3] Like in [125], determinism can be relaxed here to predictability – the fact that the environment is always informed of the choices made inside the program.

of the reactions $(\texttt{I1}^{(1,2)}, \texttt{O1}^3, \texttt{SYNC}^\bullet, \texttt{C}^0)$ or $(\texttt{I1}^{(1,2)}, \texttt{O1}^3)$ can be executed by ADD1, leading to divergent computations. The problem is that these two reactions are not independent, but no value of a given channel allows to differentiate one from the other (so one can't deterministically choose between them in an asynchronous environment).

Deterministic input event reconstruction is therefore impossible for some synchronous programs. Therefore, a methodology to implement synchronous programs on an asynchronous architecture must rely on the (implicit or explicit) identification of some class of programs for which reconstruction is possible. Then, giving a deterministic asynchronous implementation to any given synchronous program is done in two steps:

Step 1. Transforming the initial program, through added synchronizations and/or signals, so that it belongs to the implementable class.

Step 2. Generating an implementation for the transformed program.

The choice of the class of implementable programs is therefore essential. On one hand, choosing a small class can highly simplify analysis and code generation in step (2). On the other, small classes of programs result in heavier synchronization added to the programs in step (1). Our choice, justified in the next section, is the class of weakly endochronous programs.

3.2.1 Previous work

Aside from weak endochrony, the most developed notions identifying classes of implementable programs are the *latency-insensitive systems* of Carloni *et al.* [39] and the *endochronous systems* of Benveniste *et al.* [17, 77].

Latency-insensitive systems are those featuring no signal absence. Transforming processes featuring absence, such as our example of Figures 3.1 and 3.2, into latency-insensitive ones amounts to adding supplementary Boolean signals that transmit at each reaction the status of every other signal. This is easy to check and implement, but often results in an unneeded communication overhead due to messages that need to be sent at each reaction. Several variations and hardware implementations of the theory have been proposed, of which we mention here only the one by Vijayaraghavan and Arvind [149].

The endochronous systems and the related hardware-centric *generalized latency-insensitive systems* [145] are those where the presence and absence of all signals can be incrementally inferred starting from the state and from signals that are always present. For instance, Fig. 3.5 presents a run of an endochronous program obtained by transforming the SYNC signal of our example into one that carries values from 0 to 3: 0 for ADD1 executing alone, 1 for ADD2 executing alone, 2 for both adders executing without communicating (C absent), and 3 for the synchronized execution of the two adders (C present). Note that the value of SYNC determines the presence/absence of all signals.

Checking endochrony consists in ordering the signals of the process in a tree representing the incremental process used to infer signal presence (the signals

Clock	1	2	3	4	5
I1	(1,2)	(9,9)	(9,9)	(2,5)	∗
O1	3	8	8	7	∗
SYNC	0	3	2	3	1
C	∗	1	∗	0	∗
I2	∗	(0,0)	(0,0)	(1,4)	(2,3)
O2	∗	1	0	5	5

Figure 3.5: Endochronous solution

that are always read are all placed in the tree root). The compilation of the Signal/Polychrony language is currently founded on a version of endochrony [4].

The endochronous reaction reconstruction process is fully deterministic, and the presence of all signals is synchronized with respect to some base signal(s) in a hierarchic fashion. This means that no concurrency remains between subprograms of an endochronous program. For instance, in the endochronous model of our adder, the behavior of the two adders is synchronized at all instants by the `SYNC` signal (whereas in the initial model the adders can function independently whenever `SYNC` is absent). By consequence, using endochrony as the basis for the development of systems with internal concurrency has 2 drawbacks:

- Endochrony is non-compositional (synchronization code must be added even when composing programs sharing no signal).
- Specifications and implementations/simulations are often over-synchronized.

3.3 Contribution

3.3.1 Definition of weak endochrony

My first contribution here was the definition of weak endochrony, defined in collaboration with B. Caillaud and A. Benveniste [128, 127]. Weak endochrony generalizes endochrony by allowing both synchronized and non-synchronized (independent) computations to be realized by a given program. Weak endochrony determines that compound reactions that are apparently synchronous can be split into independent smaller reactions that are asynchronously feasible in a *confluent* way, so that the first one does not discard the second.

Fig. 3.6 presents a run of a weakly endochronous system obtained by replacing the `SYNC` signal of our example with two input signals:

- `SYNC1`, of Boolean type, is received at each execution of `ADD1`. It has value 0 to notify that no synchronization is necessary, and value 1 to notify that synchronization is necessary and the carry signal `C` must be produced.

- `SYNC2`, of Boolean type, is received at each execution of `ADD2`. It has value 0 to notify that no synchronization is necessary, and value 1 to notify that synchronization is necessary and the carry signal `C` must be read.

The two adders are synchronized when `SYNC1`=1 and `SYNC2`=1, corresponding to the cases where `SYNC`=• in the original design. However, the adders function independently elsewhere (between synchronization points).

I1	(1,2)	(9,9)	(9,9)		(2,5)	
O1	3	8	8		7	
SYNC1	0	1	0		1	
C		1			0	
SYNC2		1		0	1	0
I2		(0,0)		(0,0)	(1,4)	(2,3)
O2		1		0	5	5

Figure 3.6: Weakly endochronous solution.

From a practical point of view, weak endochrony supports less synchronized, concurrent GALS implementations. While the implementation of latency-insensitive and endochronous synchronous programs is strictly bound by the scheme of Fig. 3.3, wrappers of weakly endochronous programs may exploit the concurrency of the specification by directly and concurrently activating various parts of a program. In the context of the example of Fig. 3.6, the GALS wrapper may consist of an AFSM that can independently activate the two adders, the activations being synchronized only when `SYNC1`=`SYNC2`=1.

Weak endochrony provides an important theoretical tool in the analysis of concurrent synchronous systems. It generalizes to a synchronous setting [128] the theory of Mazurkiewicz traces [54]. Although it deals with the signal values[4], (weak) endochrony is in essence strongly related with the notion of *conflict-freeness*, first introduced in the context of Petri Nets, which simply states that once enabled, an action cannot be disabled, and must eventually be executed. Various conflict-free variants of data-flow declarative formalisms form the area of *process networks* (such as Kahn Process Networks [91]), or various so-called domains of the Ptolemy environment such as SDF Process Networks [30]. Conflict-freeness is also called *confluence* ("diamond property") in process algebra theory [110], and *monotony* in Kahn Process Networks.

3.3.2 Characterization of delay-insensitive synchronous components

While weak endochrony provided a *sufficient* property ensuring delay-insensitivity, exactly characterizing the class of synchronous components that can function as

[4]Which may be used to decide which further signals are to be present next causally in the reaction

delay-insensitive deterministic systems remained an open problem. I have characterized this class [120, 129]. The characterization is given by a simple diamond closure property, very close to weak endochrony. This simple characterization is important because:

- It offers a good basis for the development of optimal synchronization protocols.
- It corresponds to a very general execution mechanism covering current practice in embedded system design. Thus, it fixes theoretical limits to what can be done in practice.

3.3.3 Synthesis of delay-insensitive concurrent implementations

In [134, 131] I proposed a method to check weak endochrony on multi-clock synchronous programs. This method is based on the construction of so-called generator sets. Generators are minimal synchronization patterns of a program, and the set of all generators provides a representation of all synchronization configurations of a synchronous program. I have proposed a compact representation for generator sets and algorithms for their modular construction for Signal/Polychrony [77] programs, starting from those of the language primitives. The generator set of a program can be analyzed to determine if the specification is weakly endochronous. In case the specification is not weakly endochronous, the generator sets can be used to produce intuitive error messages helping the programmer add supplementary synchronization to the program interface (or minimal synchronization can be automatically synthesized). The algorithms have been implemented in a tool connected to the Signal/Polychrony toolset.

In case the specification is weakly endochronous, we provided a technique for building multi-threaded and distributed implementation code [118]. This technique takes advantage of the structure of the generator set to limit the number of threads, and thus reduce scheduler overhead.

Chapter 4

Reconciling performance and predictability on a many-core

The previous chapter described work on a particular aspect of the embedded implementation flow: the synthesis of efficient synchronization protocols. I will now take the first step towards considering the whole complexity of synthesizing full real-time embedded implementations. For this first step, I consider a mapping and code generation approach that is very similar to that of classical compilation. Like in classical compiler such as GCC, our scheduling routines have as objective the optimization of simple metrics, such as reaction latency or throughput. Furthermore, scheduling always succeeds if execution on the platform is functionally possible, because no real-time requirements are taken into account.

But there are also significant differences with respect to classical compilers:

- Our systems compiler performs *precise and conservative timing accounting*, which allows it to provide tight, hard real-time guarantees on the generated code. Such guarantees can then be checked against real-time requirements.
- The optimization metrics may be different from those used in classical compilation.
- Architecture descriptions used by the scheduling algorithms (and by consequence the architecture-dependent scheduling heuristics) are significantly different.

These differences mean that my work had to focus on two main aspects:

- Modeling the execution platform and ensuring that the platform models are conservative abstractions of the actual execution platform. To provide

tight and hard real-time guarantees, these models must be precise and include timing aspects.[1]

- The definition of the scheduling and code generation algorithms.

As a test case I use here a many-core platform, showing how its architectural detail can be efficiently taken into account, which is an important subject *per se*.

4.1 Motivation

One crucial problem in real-time scheduling is that of ensuring that the application software and the implementation platform satisfy the hypotheses allowing the application of specific *schedulability analysis* techniques [102]. Such hypotheses are the availability of a priority-driven scheduler or the possibility of including scheduler-related costs in the durations of the tasks. Classical work on real-time scheduling [52, 70, 66] has proposed formal models allowing schedulability analysis in classical mono-processor and distributed settings. But the advent of *multiprocessor systems-on-chip (MPSoC)* architectures imposes significant changes to these models and to the scheduling techniques using them.

MPSoCs are becoming prevalent in both general-purpose and embedded systems. Their adoption is driven by scalable performance arguments (concerning speed, power, *etc.*), but this scalability comes at the price of increased complexity of both the software and the software mapping (allocation and scheduling) process.

Part of this complexity can be attributed to the steady increase in the *quantity* of software that is run by a single system. But there are also significant *qualitative* changes concerning both the software and the hardware. In software, more and more applications include *parallel* versions of classical signal or image processing algorithms [150, 9, 69], which are best modeled using data-flow models (as opposed to *independent tasks*). Providing functional and real-time correctness guarantees for parallel code requires an accurate control of the interferences due to concurrent use of communication resources. Depending on the hardware and software architecture, this can be very difficult [154, 87].

Significant changes also concern the execution platforms, where the gains predicted by Moore's law no longer translate into improved single-processor performance, but in a rapid increase of the number of processor cores placed on a single chip [26]. This trend is best illustrated by the *massively parallel processor arrays (MPPAs)*, which are the object of the work presented in this chapter. MPPAs are MPSoCs characterized by:

- Large numbers of processing cores, ranging in current silicon implementations from a few tens to a few hundreds [148, 113, 1, 62]. The cores are typically chosen for their area or energy efficiency instead of raw computing power.

[1] These models can be seen as the equivalent of the ISAs, ABIs and APIs used in classical compilation.

- A regular internal structure where processor cores are divided among a set of identical *tiles*, which are connected through one or more NoCs with regular structure (*e.g.* torus, mesh).

Industrial [148, 113, 1, 62] and academic [71, 23, 144] MPPA architectures targeting hard real-time applications already exist, but the problem of mapping applications on them remains largely open. There are two main reasons to this. The first one concerns the NoCs: as the tasks are more tightly coupled and the number of resources in the system increases, the on-chip networks become critical resources, which need to be explicitly considered and managed during real-time scheduling. Recent work [144, 93, 115] has determined that NoCs have distinctive traits requiring significant changes to classical real-time scheduling theory [70]. In particular, they have large numbers of potential contention points, limited buffering capacities requiring synchronized resource reservation, and network control often operates at the level of small data packets. The second reason concerns automation: the complexity of MPPAs and of the (parallel) applications mapped on them is such that the *allocation and scheduling must be largely automated*.

Unlike previous work on the subject I have addressed these needs by relying on static (off-line) scheduling approaches. In theory, off-line algorithms allow the computation of *scheduling tables* specifying an *optimal* allocation and real-time scheduling of the various computations and communications onto the resources of the MPPA. In practice, this ability is severely limited by 3 factors:

1. The **application** may exhibit a high degree of dynamicity due to either environment variability or to execution time variability resulting from data-dependent conditional control.[2]

2. The **hardware** may not allow the implementation of optimal scheduling tables. For instance, most MPPA architectures provide only limited control over the scheduling of communications inside the NoC.

3. The **mapping** problems I consider are NP-hard. In practice, this means that optimality cannot be attained, and that efficient heuristics are needed.

Clearly, not all applications can benefit from off-line scheduling. But this paradigm is well adapted to our target application class: (parallelized versions of) periodic embedded control and signal processing applications. Our work shows that for such applications, off-line scheduling techniques attain both high timing predictability and high performance. But reconciling performance and predictability (two properties often seen as antagonistic) was only possible by considering with a unifying view all the hardware, software, and mapping aspects of the design flow.

My contributions concern all these aspects, which will be covered one by one in the following sections. On the hardware side, an in-depth review of

[2]Implementing an optimal control scheme for such an application may require more resources than the application itself, which is why dynamic/on-line scheduling techniques are often preferred.

MPPA/NoC architectures with support for real-time scheduling allowed us to determine that NoCs allowing static communication scheduling offer the best support to off-line application mapping (and I have participated in the definition of such a NoC and MPPA based on it). I have also proposed a software organization that improves timing predictability. These hardware and software advances allowed the definition of a technique for the WCET analysis of parallel code.

But my main effort went into defining a new technique and tool, called LoPhT, for automatic real-time mapping and code generation, whose global flow is pictured in Fig. 4.1. Our tool takes as input data-flow synchronous specifica-

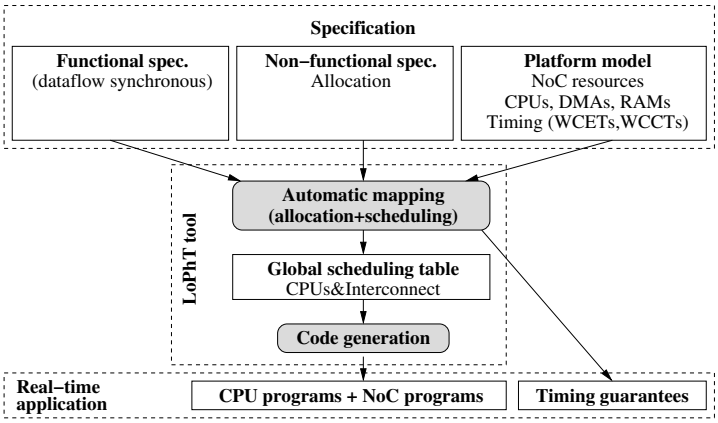

Figure 4.1: Global flow of the proposed mapping technique for many-cores

tions and precise hardware descriptions including all potential NoC contention points. It uses advanced off-line scheduling techniques such as software pipelining and pre-computed preemption, and it takes into account the specificities of the MPPA hardware to build *scheduling tables* that provide good latency and throughput guarantees and ensure an efficient use of computation and communication resources. Scheduling tables are then automatically converted into sequential code ensuring the correct ordering of operations on each resource and the respect of the real-time guarantees. *Experimental evaluations show that the off-line mapping of communications not only allows us to provide static latency and throughput guarantees, but may also improve the speed of the application, when compared to (simple) hand-written parallel code.*

Through these results, I have shown that *taking into account the fine detail of the hardware and software architecture of a many-core is possible, and allows off-line mapping of very good precision*, even when low-complexity scheduling heuristics are used in order to ensure the scalability of the approach. This is similar to what compilation did to replace manual assembly coding.

4.2 MPPA/NoC architectures for the real-time

This section starts with a general introduction to MPPA platforms, and then presents the main characteristics of existing NoCs, with a focus on flow management mechanisms supporting real-time implementation.

4.2.1 Structure of an MPPA

My work concerns hard real-time systems where timing guarantees must be determined by static analysis methods before system execution. Complex memory hierarchies involving multiple cache levels and cache coherency mechanisms are known to complicate timing analysis [154, 80], and I assume they are not used in the MPPA platforms I consider. Under this hypothesis, *all* data transfers between tiles are performed through one or more NoCs.

A NoC can be described in terms of point-to-point communication links and NoC routers which perform the routing and scheduling (arbitration) functions. Fig. 4.2 provides the description of a 2-dimensional (2D) mesh NoC like the ones used in the Adapteva Epiphany [1], Tilera TilePro[148], or DSPIN[117]. The structure of a router in a 2D mesh NoC is described in Fig. 4.3. It has 5 connec-

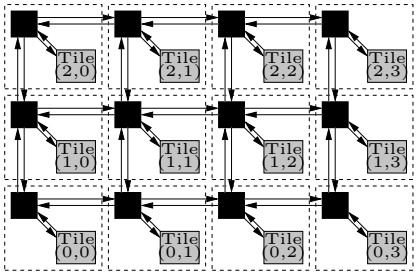

Figure 4.2: An MPPA platform with 4x3 tiles connected with a 2D mesh NoC. Black rectanges are the NoC routers. Tile coordinates are in (Y,X) order.

tions (labeled North, South, West, East, and Local) to the 4 routers next to it and to the local tile. Each connection is formed of a routing component, which we call *demultiplexer* (labeled **D** in Fig. 4.3), and a scheduling/arbitration component, which we call *multiplexer* (labeled **M**). Data enters the router through demultiplexers and exits through multiplexers.

To allow on-chip implementation at a reasonable area cost, NoCs use simple routing algorithms. For instance, the Adapteva [1], Tilera [148], and DSPIN NoCs [117] use an X-first routing algorithm where data first travels all the way in the X direction, and only then in the Y direction. Furthermore, all NoCs mentioned in this chapter use simple *wormhole* switching approaches [114] requiring that all data of a communication unit (*e.g.* packet) follow the same route, in order, and that the communication unit is not logically split during transmission.

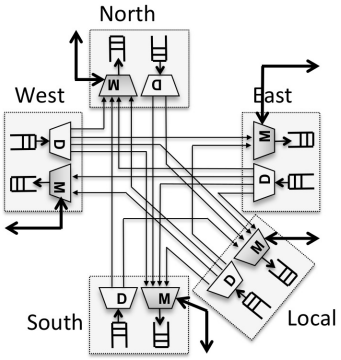

Figure 4.3: Generic router for a 2D mesh NoC with X-first routing policy

The use of a wormhole switching approach is justified by the limited buffering capabilities of NoCs [115] and by the possibility of decreasing transmission latencies (by comparison with more classical store-and-forward approaches). But the use of wormhole switching means that one data transmission unit (such as a packet) is seldom stored in a single router buffer. Instead, a packet usually spans over several routers, so that its *transmission strongly synchronizes multiplexers and demultiplexers along its path.*

4.2.2 Support for real-time implementation

Given the large number of potential contention points (router multiplexers), and the synchronizations induced by data transmissions, providing *tight* static timing guarantees is only possible if some form of flow control mechanism is used.

In NoCs based on *circuit switching* [88], inter-tile communications are performed through dedicated *communication channels* formed of point-to-point physical links. Two channels cannot share a physical link. This is achieved by statically fixing the output direction of each demultiplexer and the data source of each multiplexer along the channel path. Timing interferences between channels are impossible, which radically simplifies timing analysis, and the latency of communications is low. But the absence of resource sharing is also the main drawback of circuit switching, resulting in low numbers of possible communication channels and low utilization of the NoC resources. Reconfiguration is usually possible, but it carries a large timing penalty.

Virtual circuit switching is an evolution of circuit switching which allows resource sharing between circuits. But resource sharing implies the need for arbitration mechanisms inside NoC multiplexers. Very interesting from the point of view of timing predictability are NoCs where arbitration is based on time division multiplexing (TDM), such as Aethereal [71], Nostrum [109], and others [147]. In a TDM NoC, all routers share a common time base. The point-

to-point links are reserved for the use of the virtual circuits following a fixed cyclic schedule (a scheduling table). The reservations made on the various links ensure that communications can follow their path without waiting. TDM-based NoCs allow the computation of *precise latency and throughput guarantees*. They also ensure a strong *temporal isolation* between virtual circuits, so that changes to a virtual circuit do not modify the real-time characteristics of the other.

When no global time base exists, the same type of latency and throughput guarantees can be obtained in NoCs relying on *bandwidth management* mechanisms such as Kalray MPPA [113, 83]. The idea here is to ensure that the throughput of each virtual circuit is limited to a fraction of the transmission capacity of a physical point-to-point link, by either the emitting tile or by the NoC routers. Two or more virtual circuits can share a point-to-point link if their combined transmission needs are less than what the physical link provides.

But TDM and bandwidth management NoCs have certain limitations: One of them is that latency and throughput are correlated [144], which may result in a waste of resources. But the latency-throughput correlation is just one consequence of a more profound limitation: TDM and bandwith management NoCs largely ignore the fact that the needs of an application may change during execution, depending on its *state*. For instance, when scheduling a data-flow synchronous graph with the objective of reducing the duration of one computation cycle (also known as *makespan* or *latency*), it is often useful to allow some communications to use 100% of the physical link, so that they complete faster, before allowing all other communications to be performed.

One way of taking into account the application state is by using NoCs with support for *priority-based scheduling* [144, 117, 62]. In these NoCs, each data packet is assigned a priority level (a small integer), and NoC routers allow higher-priority packets to pass before lower-priority packets. To avoid *priority inversion* phenomenons, higher-priority packets have the right to preempt the transmission of lower-priority ones. In turn, this requires the use of one separate buffer for each priority level in each router multiplexer, a mechanism known as *virtual channels (VCs)* in the NoC community[117].

The need for VCs is the main limiting factor of priority-based arbitration in NoCs. Indeed, adding a VC is as complex as adding a whole new NoC[157, 33], and NoC resources (especially buffers) are expensive in both power consumption and area [112]. Among existing silicon implementations only the Intel SCC chip offers a relatively large numbers of VCs (eight) [62], and it is targeted at high-performance computing applications. Industrial MPPA chips targeting an embedded market usually feature multiple, specialized NoCs [148, 1, 83] without virtual channels. Other NoC architectures feature low numbers of VCs. Current research on priority-based communication scheduling has already integrated this limitation, by investigating the sharing of priority levels [144].

Significant work already exists on the mapping of real-time applications onto priority-based NoCs [144, 138, 115, 93]. This work has shown that priority-based NoCs support the efficient mapping of *independent* tasks.

But we already explained that the large number of computing cores in an MPPA means that applications are also likely to include parallelized code which

is best modeled by large sets of relatively small dependent tasks (data-flow graphs) with predictable functional and temporal behavior [150, 9, 69]. Such timing-predictable specifications are those that can *a priori* take advantage of a static scheduling approach, which provides best results on architectures with support for static communication scheduling [148, 61, 55]. Such architectures allow the construction of an efficient (possibly optimal) global computation and communication schedule, represented with a scheduling table and implemented as a set of synchronized *sequential computation and communication programs*. Computation programs run on processor cores to sequence task executions and the initiation of communications. Communication programs run on specially-designed micro-controllers that control each NoC multiplexer to fix the order in which individual data packets are transmitted. Synchronization between the programs is ensured by the data packet communications themselves.

Like in TDM NoCs, the use of global computation and communication scheduling tables allows the computation of very precise latency and throughput estimations. Unlike in TDM NoCs, NoC resource reservations can depend on the application state. Global time synchronization is not needed, and existing NoCs based on static communication scheduling do not use it[148, 61, 55]. Instead, global synchronization is realized by the data transmissions (which eliminates some of the run-time pessimism of TDM-based approaches).

The microcontrollers that drive each NoC router multiplexer are similar in structure to those used in TDM NoCs to enforce the TDM reservation pattern. The main difference is that the communication programs are usually longer than the TDM configurations, because they must cover longer execution patterns. This requires the use of larger program memory (which can be seen as part of the tile program memory[55]). But like in TDM NoCs, buffering needs are limited and no virtual channel mechanism is needed, which results in lower power consumption.

From a mapping-oriented point of view, determining exact packet transmission orders cannot be separated from the larger problem of building a global scheduling table comprising both computations and communications. By comparison, mapping onto MPPAs with TDM-based or bandwith reservation-based NoCs usually separates task allocation and scheduling from the synthesis of a NoC configuration independent from the application state [104, 9, 160].

Under static communication scheduling, there is little run-time flexibility, as all scheduling possibilities must be considered during the off-line construction of the global scheduling table. For very dynamic applications this can be difficult. This is why existing MPPA architectures that allow static communication scheduling also allow communications with dynamic (Round Robin) arbitration.

In conclusion, NoCs allowing static communication scheduling offer the best temporal precision in the off-line mapping of dependent tasks (data-flow graphs), while priority-based NoCs are better at dealing with more dynamic applications. As future systems will include both statically parallelized code and more dynamic aspects, NoCs should include mechanisms supporting both off-line and on-line communication scheduling. Significant work already exists on the real-time mapping for priority-based platforms, while little work has addressed the

NoCs with static communication scheduling – the subject I addressed with my PhD students and post-docs and detailed in this chapter.

4.2.3 MPPA platform for off-line scheduling

Tiled many-cores in SoCLib

The SoCLib virtual prototyping library [101] allows the definition of tiled many-cores following a *distributed shared memory* paradigm where all memory banks and component programming interfaces are assigned unique addresses in a global address space. All memory transfers and component programming operations are represented with memory accesses organised as command/response transactions according to the VCI/OCP protocol [151]. To avoid interferences between commands and responses (which can lead to deadlocks), the on-chip interconnect is organized in two completely disjoint sub-networks, one for transmitting commands, and the other for responses.

There are two types of transactions: read and write. In write transactions the command sub-network carries the data to be written and the target adress, and the response sub-network carries a return code. In read transactions, the command sub-network carries the address and size of the requested data, and the response sub-network carries the data.

The tiled many-cores of SoCLib are formed of a rectangular set of tiles connected through a state-of-the-art 2D synchronous mesh network-on-chip (NoC) called DSPIN [117]. The NoC is formed of a command NoC and a response NoC which are fully separated. Note that the representation of Fig. 4.2 is incomplete, as it only represents one of the NoCs. However, we shall see in the following sections that it is sufficient to allow scheduling under certain hypotheses. Each tile has its own command and response local interconnects, linked to the NoCs and to the IP cores of the tile (CPUs, RAMs, DMAs, etc.).

Modifications of the tile structure

To improve timing predictability and worst-case performance, we modify both the tiles and the NoC of the SoCLib-based many-core. Of the original organization, we retain the global organization of the many-core, and in particular its distributed shared memory model which allows programming using general-purpose tools. Fig. 4.4 pictures the structure of the computing tile in the original SoCLib many-core, and Fig. 4.5 its modified version.

The memory subsystem. Our objective here is to improve timing predictability by eliminating contentions. In our experiments with the original SoCLib-based many-core, the second most important source of contentions (after the NoC) is the access to the unique RAM bank of each tile. To reduce these contentions, we decided to follow the example of existing industrial many-core architectures [108, 83], and replace the single RAM bank of a tile with several memory banks that can be accessed independently.

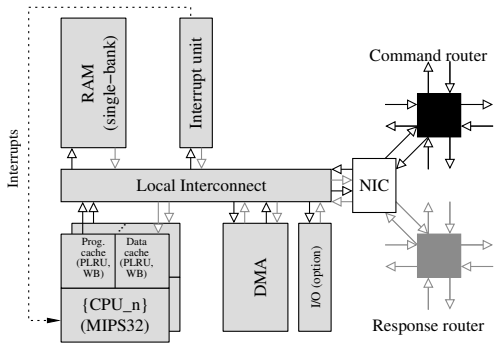

Figure 4.4: The computing tile in the original DSPIN-based many-core architecture

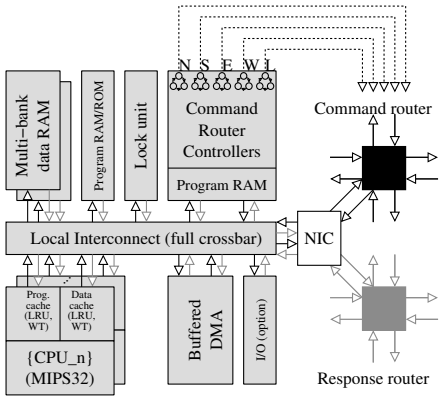

Figure 4.5: Modified computing tile of our architecture

To facilitate timing analysis, we separate data (including stack) and program memory. One RAM bank is used in each tile to store the program of all the CPUs of the tile. Data and stack are stored on a *multi-bank RAM*. Each bank of the data RAM has a separate connection to the local interconnect. RAM banks of a tile are assigned contiguous address ranges, so that they can store data structures larger than a single tile. Explicit allocation of data onto the memory banks, along with the use of lock-based synchronization and the topology of the local interconnect presented below allow the elimination of contentions due to concurrent access to memory banks, by ensuring that no data bank is accessed from two sources (CPUs or DMAs) at the same time.

Note that the use of a multi-bank data RAM also removes a significant performance bottleneck of the original architecture. Indeed, a single RAM bank can only serve 4 CPUs. Experimental data shows that placing more than 4

CPUs per tile results in no performance gain because the RAM access is saturated. Having multiple RAM banks per tile removes this limitation. Our test configurations use a maximum of 16 CPU cores per tile and two data RAM banks per CPU core, for a maximum of 4Mbytes of RAM per tile.

The local interconnect is chosen in our design so that it cannot introduce contentions due to its internal organization. Contentions can still happen, for instance, when two CPUs access concurrently the program memory. However, accesses from different sources to different targets never introduce contentions. Interconnect types allowing this are the full crossbars and the multi-stage interconnection network [12] such as the omega networks, the delta networks, or the related logarithmic interconnect [92]. My experiments used a full crossbar interconnect.

The CPU core we use is a single-issue, in-order, pipelined implementation of the MIPS32 ISA with no speculative execution. We did not change this, as it simplifies timing analysis and allows small-area hardware implementation. However, significant work has been invested in designing a cycle-accurate model of this core inside a state-of-the art WCET analysis tool [133].

The caches have been significantly modified. The original design featured caches with a pseudo-LRU (PLRU) replacement policy and with a writing policy that is intermediate between write-through and write-back.[3] Memory accesses from the data and instruction caches of a single CPU were multiplexed over a single conection to the local interconnect of the tile. All these choices are known to complicate timing analysis and/or to reduce the precision of the analysis [137, 80], and thus we reverted to more conservative choices: We use the LRU replacement policy, a fully write-through policy, and we let the instruction and data caches access the local tile interconnect through separate connexions. The use of a write-through policy reduces the processing speed of each CPU. This is the only modification we made on the architecture that decreases processing speed.

Synchronization. To improve temporal predictability, and also speed, our architecture does not use interrupt-based synchronization. Interrupt signaling by itself is fast, but handling an interrupt usually requires accesses to program memory which take supplementary time. Furthermore, arrival date imprecision and modifications of the cache state mean that interrupts are difficult to take into account during timing analysis. To avoid these performance and predictability problems, we replace the interrupt unit present in each tile of the original architecture with a hardware lock component. These components allow synchronization with very low overhead (1 non-cached local RAM access) and

[3] Consecutive writes inside a single cache ligne were buffered.

without modifications of the cache state. The lock unit follows a simple request/grant protocol which allows a single grant operation to unblock multiple requests.

Buffered DMA. The traditional DMA unit used in the original architecture requires significant software control to determine when a DMA operation is finished so that another can start. This is either done using interrupt-based signaling, which has the inconvenients mentioned above, or through polling of the DMA registers, which requires significant CPU time and imposes significant constraints on CPU scheduling.

To avoid these problems, we use DMA units allowing the buffering of transmission commands. A CPU can send one or more DMA commands while the previous DMA operation is not yet completed. Furthermore, the DMA unit can be programmed so that it not only sends out data, but also signals the end of the transmission to the target tile by granting a lock, as described in Section 4.3. Thus, all inter-tile communication and synchronization can be performed by the DMA units, in parallel with the data computations of the CPUs and without requiring significant CPU time for control.

Modifications of the NoC

The DSPIN network-on-chip [117] is a classical 2D mesh NoC. It uses wormhole packet switching and a static routing scheme[4]. Each router of the command or response NoC has the internal structure of Fig. 4.3. Each NoC router is connected through FIFO links with the 4 neighboring routers (denoted with North, South, West, East) and with the local computing tile. Each of these connections is realized through one demultiplexer and one multiplexer. The demultiplexer ensures the routing function (X-first/Y-first). It reads the headers of the incoming packets and, depending on the target address, sends them towards one of the multiplexers of the router. The multiplexer ensures the arbitration (scheduling) function. When two packets arrive at the same time (from different demultiplexers), a fair Round Robin arbiter is used to decide which one will be transmitted first. Once the transmission of a packet is started, it cannot be stopped.[5]

The fair arbitration scheme is well-adapted to applications without real-time requirements, where it ensures a good use of NoC resources. But when the objective is to provide real-time guarantees and to allocate NoC resources according to application needs, it is better to use some other arbitration mechanism. In our case, the objective is to provide the best possible support for the implementation of static computation and communication schedules. Therefore, we rely on a *programmed arbitration* approach where each router multiplexer enforces a fixed packet transmission order specified under the form of a *communication program*.

[4] X-first routing for the command network and Y-first for the response network.

[5] Unless a virtual channel mechanism is used, as described in [117].

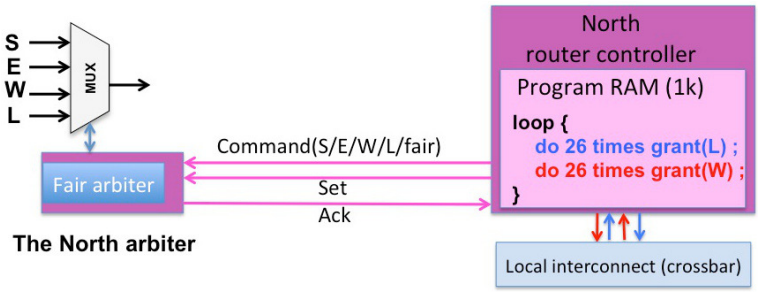

Figure 4.6: Programmed arbitration in a NoC router multiplexer

Enforcing fixed packet transmission orders requires the use of new hardware components called *router controllers*. These components are present in the tile description of Fig. 4.5 and their interaction with the NoC router multiplexers is realized as pictured in Fig. 4.6. Each multiplexer of the command network is controlled by its own controller running a separate program. The program is loaded onto the component through the local interconnect. The interface between router controller and the local interconnect is also used to start or stop the router controller. When the controller is stopped, the fair arbiter of DSPIN takes control. In Fig. 4.6, the arbitration program will cyclically accept 26 packets from the Local connection, then 26 packets from the West connection.

More details on the implementation and properties of our programmed arbitration mechanism can be found in [55].

4.3 Software organization

On the hardware architecture defined above, we use non-preemptive, statically-scheduled software with lock-based inter-processor synchronization. Each core is allocated one sequential thread. To ensure both performance and predictability, we require that software follows the organization rules detailed in this section.

Data locality. We require computation functions to only operate on local tile data. If a computation needs data from another tile, this data is first transferred to the local data RAM. Under this hypothesis, the timing (WCET/BCET) analysis of computation functions can be performed without considering the NoC.

Inter-tile data transfers. They are only performed using the DMA units. The CPUs still retain the possibility of directly accessing RAM banks of other tiles, but they only do so during the boot phase (which follows the standard protocol of the MIPS32 ISA), or for non-real-time code running on many-core tiles allocated to such code. Traffic generated directly by CPUs and their caches

has very small grain (usually a single data word per memory write access), and it is difficult to accurately predict its timing. Thus, not allowing it to traverse the NoC largely simplifies the timing analysis of both NoC transfers and CPU code [154].

Inter-tile data transfers and synchronizations are only performed through write transactions performed by the DMA unit of the sending tile. Thus, the response NoC only carries 2-flit acknowledge packets, so that contentions on the response NoC are negligible even in the absence of programmed arbitration. This is why router controllers are only used for the command NoC multiplexers, leaving unchanged the fair arbiters on the response network.

Allocation of the tile memory. The memory allocation scheme we used for automatic code generation and for the case studies makes several assumptions. First, we assume that the programs of all CPUs in a tile are stored in the local program memory. This amounts to either assuming that this memory is a non-volatile one, or that program loads outside the boot phase are explicitly scheduled as data transfers over the NoC.

Second, we allocate one of the data RAM banks for the stacks of all the CPUs of the tile. Using only one RAM bank for all the stacks is possible because our applications only make little use of the stack (most data is explicitly allocated by our tool on the other memory banks).

Allocating all programs (respectively all stacks) on a single memory bank means that the cost of a cache miss due to a program (resp. stack) memory access can be very high, due to interference from the other CPU cores of the tile. However, the (relatively) small size of the programs (resp. stacks) means that misses seldom occur, so that the high cost of a miss will not result in overly pessimistic WCET estimations. For applications with large programs or with significant use of the stack, other memory allocation approaches can be used.

All data RAM banks except the stack-dedicated one are allocated to data variables. To each data variable we associate a contiguous memory region with statically-defined start address and length. The length of the region must be at least equal to the worst-case size of the data type of the variable (which must be computed during code generation).

4.4 WCET analysis of parallel code

Classical timing analysis techniques for parallel code isolate micro-architecture analysis from the analysis of synchronizations between cores by performing them in two separate analysis phases (WCET and WCRT analysis). This isolation has its advantages, such as a reduction of the complexity of each analysis phase, and a separation of concerns that facilitates the development of analysis tools. But isolation also has a major drawback: a loss in precision which can be significant. To consider only one aspect, in order to be safe the WCET analysis of each synchronization-free sequential code region has to consider an undetermined initial micro-architecture state (of caches and pipeline). This may result in

overestimated WCETs, and consequently in pessimistic execution time bounds for the whole parallel application.

My contribution on this subject (in collaboration with I. Puaut) [133] is an *integrated* WCET analysis approach that considers at the same time micro-architectural information and the synchronizations between cores. This is achieved by extending a state-of-the-art WCET estimation technique and tool to manage synchronizations and communications between the sequential threads running on the different cores. The benefits of the proposed method are twofold. On the one hand, the micro-architectural state is not lost between synchronization-free code regions running on the same core, which results in tighter execution time estimates. On the other hand, only one tool is required for the temporal validation of the parallel application, which reduces the complexity of the timing validation toolchain.

Such a holistic approach is made possible by the use of the deterministic and composable software and hardware architectures defined in the previous sections. Experimental results show that the integrated approach always produces better WCET estimations (21% precision gain on our test applications) and that these estimations are close to measured execution time.

4.5 Mapping (1) - MPPA-specific aspects

4.5.1 Resource modeling

To allow off-line mapping onto our architectures, we need to identify the set of *abstract* computation and communication resources that are considered during allocation and scheduling.

We associate one communication resource to each of the multiplexers of NoC routers and to each DMA. We name them as follows: $N(i,j)(k,l)$ is the inter-router wire going from $Tile(i,j)$ to $Tile(k,l)$; $In(i,j)$ is the output of the router (i,j) to the local tile; $DMA(i,j)$ is the output of $Tile(i,j)$ to the local router.

My work has mostly focused on NoC modeling and handling. To this end, I considered a resource model that simplifies as much as possible the representation of the computing tiles. All the 16 processor cores of the tile are seen as a single, very fast computing resource. This means that operations will be allocated to the tile as if it were a sequential processor, but the allocated operations are in fact parallel code running on all 16 processors.

This means that Fig. 4.2 includes all the resources of my platform model. There are just 12 tile resources representing 192 processor cores, and the 58 arcs represent the NoC resources (of the command network).

Communication durations

All inter-tile data transmissions are performed using the DMA units. If a transmission is not blocked on the NoC, then its duration on the sender side only depends on the size of the transmitted data. The exact formula is $d = s + \lceil s/\underline{MaxPayload} \rceil * \underline{PacketHeaderSize}$, where d is the duration in clock cycles

of the DMA transfer from the start of the transmission to the cycle where a new transmission can start, s is the data size in 32-bit words, $\underline{MaxPayload}$ is the maximum payload of a NoC packet (in 32-bit words), and $\underline{PacketHeaderSize}$ is the number of clock cycles needed to transmit the packet header. In our case, $\underline{MaxPayload}$=16 flits and $\underline{PacketHeaderSize}$=4 flits.

In addition to this transmission duration, we must also account in our computations for:

- The DMA transfer initiation, which consists in 3 uncached RAM accesses plus the duration of the DMA reading the payload of the first packet from the data RAM. This cost is over-approximated as 30 cycles.

- The latency of the NoC, which is the time needed for one flit to traverse the path from source to destination. This latency is of $3 * n$, where n is the number of NoC multiplexers on the route of the transmission.

4.5.2 Application specification

The input specification of our mapping algorithms is the single-clock data-flow synchronous language Clocked Graphs, defined in [34, 130] (direct translation into this language is possible from a variant of Lustre/Scade, called Heptagon and from SynDEx). But to enable a clearer presentation of our allocation and real-time scheduling algorithms, we use a more abstract and less expressive[6] description of applications, closer in form to that of more classical task models.

Definition 1 (Non-conditioned dependent task system) *A non-conditioned dependent task system D is a directed graph with two types of arcs $D = \{T(D), A(D), \Delta(D)\}$. Here, $T(D)$ is the finite set of tasks (data-flow blocks). The finite set $A(D)$ contains dependencies of the form $a = (src(a), dst(a), type(a))$, where $src(a), dst(a) \in T(D)$ are the source, respectively the destination task of a, and $type(a)$ is the type of data transmitted from $src(a)$ to $dst(a)$. The directed graph determined by $A(D)$ must be acyclic. The finite set $\Delta(D)$ contains* delayed *dependencies of the form $\delta = (src(\delta), dst(\delta), type(\delta), depth(\delta))$, where $src(\delta)$, $dst(\delta)$, $type(\delta)$ have the same meaning as for simple dependencies and $depth(\delta)$ is a strictly positive integer called the* depth *of the dependency.*

Non-conditioned dependent task systems have a cyclic execution model. At each execution cycle of the task system, each of the tasks is executed exactly once. We denote with t^n the instance of task $t \in T(D)$ for cycle n. The execution of the tasks inside a cycle is partially ordered by the dependencies of $A(D)$. If $a \in A(D)$ then the execution of $src(a)^n$ must be finished before the start of $dst(a)^n$, for all n. Note that dependency types are explicitly defined, allowing us to manipulate communication mapping.

[6] Expressiveness loss is related to the representation of parameters used in the computation of clocks.

The dependencies of $\Delta(D)$ impose an order between tasks of successive execution cycles. If $\delta \in \Delta(D)$ then the execution of $src(\delta)^n$ must complete before the start of $dst(\delta)^{n+depth(\delta)}$, for all n.

We make the assumption that a task has no state unless it is explicitly modeled through a delayed arc. This assumption is a semantically sound way of providing more flexibility to the scheduler. Indeed, assuming by default that all tasks have an internal state (as classical task models do) implies that two instances of a task can never be executed in parallel. Our assumption does not imply restrictions on the way systems are modeled. Indeed, past and current practice in synchronous language compilation already relies on separating state from computations for each task, the latter being represented under the form of the so-called *step function* [10]. Thus, existing algorithms of classical synchronous compilers can be used to put high-level synchronous specifications into the form required by our scheduling algorithms.[7]

Definition 1 is similar to classical definitions of dependent task systems in the real-time scheduling field [47], and to definitions of data dependency graphs used in software pipelining [2, 48].

But we need to extend this definition to allow the efficient manipulation of specifications with multiple execution modes. The extension is based on the introduction of a new *exclusion relation* between tasks, as follows:

Definition 2 (Dependent task system) *A dependent task system is a tuple* $D = \{T(D), A(D), \Delta(D), EX(D)\}$ *where* $\{T(D), A(D), \Delta(D)\}$ *is a non-conditioned dependent task set and* $EX(D)$ *is an exclusion relation* $EX(D) \subseteq T(D) \times T(D) \times \mathbb{N}$.

The introduction of the exclusion relation modifies the execution model defined above as follows: if $(\tau_1, \tau_2, k) \in EX(D)$ then ${\tau_1}^n$ and ${\tau_2}^{n+k}$ are never both executed, for any execution of the modeled system and any cycle index n. For instance, if the activations of τ_1 and τ_2 are on the two branches of a test we will have $(\tau_1, \tau_2, 0) \in EX(D)$.

The relation $EX(D)$ is obtained by analysis of the clocks of the initial synchronous specification. Various algorithms have been proposed to this intent in sycnhronous language compilation and in previous work on LoPhT [34, 130]. The relation $EX(D)$ needs not be computed exactly. Any sub-set of the exact exclusion relation between tasks can safely be used during scheduling (even the void sub-set). However, the more exclusions we take into account, the better results the scheduling algorithms will give because tasks in an exclusion relation can be allocated the same resources at the same dates. This is a form of safe double allocation of resources, discussed in Section 4.6.

We exemplify our formalism with the dependent task set of Fig. 4.7, which is a simplified version of an automotive platooning application [135]. In our figure, each block is a task, solid arcs are simple dependencies, and the dashed arc is a delayed dependency of depth 2. The application is run by a car to

[7] This has already been done for a Lustre/Scade dialect [49] and for SynDEx specifications [34].

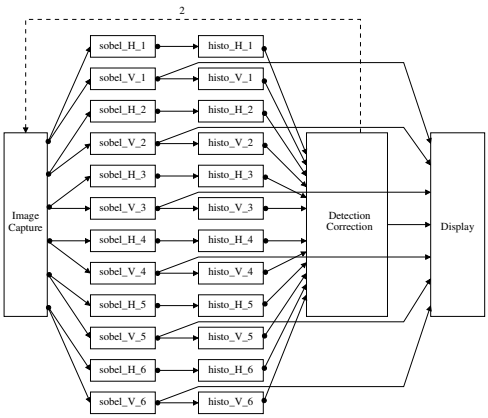

Figure 4.7: Dependent task set of a platooning application

determine the position (distance and angle) of another car moving in front of it. It works by cyclically capturing an input image of fixed size. This image is passed through an edge-detecting Sobel filter and then through a histogram search to detect dominant edges (horizontal and vertical). This information is used by the detection and correction function to determine the position of the front car. The whole process is monitored on a display. The delayed dependency represents a feedback from the detection and correction function that allows the adjustment of image capture parameters.

The Sobel filter and the histogram search are parallelized. Each of the *Sobel_H* and *Sobel_V* functions receives one sixth of the whole image (a horizontal slice).

4.5.3 Non-functional properties

For each task $\tau \in T(D)$, we define $WCET(\tau)$ to be a safe upper bound for the *worst-case execution time (WCET)* of τ on an MPPA tile, in isolation. Note that the WCET values we require are for *parallel* code running on all the 16 processors of a tile. Tight WCET bounds for such code can be computed using the analysis technique proposed in Section 4.4.

For each data type t associated with a dependency (simple or delayed), we define the worst-case memory footprint of a value of type t. This information allows the computation of the *worst-case communication time (WCCT)* for a data of that type, using the formula of Section 4.5.1.

Allocation constraints specify on which tiles a given dataflow block can be executed. In our example, they force the allocation of the capture and display functions onto specific MPPA tiles. More generally, they can be used to confine an application to part of the MPPA, leaving the other tiles free to execute other applications.

4.5.4 Scheduling and code generation

The problem

The real-time mapping and code generation problem we consider in this chapter is a bi-criteria optimization problem: We assume given an MPPA model of fixed size (fixed number of tiles, processors, memory banks, *etc.*), an application represented with a dependent task set and a non-functional specification. The problem is to synthesize a real-time implementation of the application on the MPPA that minimizes execution cycle latency (duration) and maximizes throughput[8], with priority given to latency. We chose this scheduling problem because it is meaningful in embedded systems design and because its simple definition allows us to focus on the handling of NoC-related issues.

Our allocation and scheduling problem being NP-complete, we do not aim for optimality. Instead, we rely on low-complexity heuristics that allow us to handle large numbers of resources and tasks with high temporal precision. Mapping and code generation is realized in 3 phases: The first one takes into account the dependencies of $A(D)$ in order to produce a latency-optimizing scheduling table. The second phase takes into account the delayed dependencies of $\Delta(D)$. It uses the software pipelining algorithms of [37] to improve throughput while not changing latency. Finally, once a scheduling table is computed, it is implemented in phase 3 in a way that preserves its real-time properties. We now present phases 1 and 3, while the software pipelining algorithms are presented in Section 4.6.

Phase 1: Latency-optimizing scheduling

Our scheduling routine builds a *global scheduling table covering all MPPA resources*. It uses a non-preemptive scheduling model for the tasks, and a preemptive one for the NoC communications. The reason for this is that task preemptions would introduce important temporal imprecision (through the use of interrupts), and are avoided. Data communications over the NoC are naturally divided into packets that are individually scheduled by the NoC multiplexer programs, allowing a form of pre-computed preemption with very small cost.

For each task, our scheduling routine reserves exactly one time interval on one of the tiles. For every dependency between two tasks allocated on different tiles, the scheduling routine reserves one or more time intervals on each resource along the route between the two tiles, starting with the DMA of the source tile, and continuing with the NoC multiplexers (recall that the route is fixed under the X-first routing policy).

The scheduling algorithm uses a simple list scheduling heuristic. The tasks of the dependent task system are traversed one by one in an order compatible with the dependencies between them (only the simple dependencies, not the delayed ones, which are considered during throughput optimization). Resource

[8]Throughput in this context means the number of execution cycles started per time unit. It can be larger than the inverse of the latency because we allow one cycle to start before the end of previous ones, provided that data-flow dependencies are satisfied.

allocation for a task *and for all communications associated with its input dependencies* is performed upon traversal of the task, and never changed afterwards. Scheduling starts with an empty scheduling table which is incrementally filled as the tasks and the associated communications are reserved time intervals on the various resources.

For each task, scheduling is attempted on all the tiles that can execute the block (as specified by the allocation constraints), at the earliest date possible. Among the possible allocations, we retain the one that minimizes a cost function. This cost function should be chosen so that the final length of the scheduling table is minimized (this length gives the execution cycle latency). Our choice of cost function combines the end date of the task in the schedule (with 95% weight) and the maximum occupation of a CPU in the current scheduling table (with 5% weight). Experience showed that this function produces shorter scheduling tables than the cost function based on task end date alone (as used in [76, 130]) because it reduces the scattering of computations over tiles.

Mapping NoC communications The most delicate architecture-related part of our scheduling routine for many-cores is the communication mapping function (procedure 1 in [36]). When a task is mapped on a tile, this routine is called once for each of the input dependencies of the task, if the dependency source is on another tile and if the associated data has not already been transmitted.

Procedure 1 MapCommunicationOnPath

Input: *Path* : list of resources (the communication path)
StartDate : date after which the data can be sent
DataSize : worst-case data size (in 32-bit words)
Input/Output: *SchedulingTable* : scheduling table

1: **for** $i := 1$ to ***length***(*Path*) **do**
2: *ShiftSize* := $(i-1) * \underline{SegmentBufferSize}$;
3: *FreeIntervalList*[i] :=
GetIntervalList(*SchedulingTable*, ***GetSegment***(*Path*,i), *ShiftSize*)
4: *ShiftedIntervalList*[i] :=
ShiftLeftIntervals(*FreeIntervalList*[i],*ShiftSize*)
5: *PathFreeIntervalList* :=***IntersectIntervals***(*ShiftedIntervalList*);
6: (*ReservedIntervals*,*NewIntervalList*,*NewScheduleLength*) :=
ReserveIntervals(*DataSize*,*PathFreeIntervalList*,
length(*SchedulingTable*));
7: (*IntervalForLock*,*NewIntervalList*,*NewScheduleLength*) :=
ReserveIntervals(*LockPacketLength*,*NewIntervalList*,
NewScheduleLength);
8: *ReservedIntervals* := ***AppendList***(*ReservedIntervals*,*IntervalForLock*)
9: **for** $i := 1$ to ***length***(*Path*) **do**
10: *ShiftSize* := (i-1)*$\underline{SegmentBufferSize}$;
11: *FinalIntervals*[i] :=$\overline{\textbf{\textit{ShiftRightIntervals}}}$(*ReservedIntervals*,*ShiftSize*);
12: **if** *NewScheduleLength* > ***length***(*SchedulingTable*) **then**
13: *SchedulingTable* :=
IncreaseLength(*SchedulingTable*,*NewScheduleLength*);
14: *SchedulingTable* :=
UpdateSchedulingTable(*SchedulingTable*,*Path*,*FinalIntervals*);

Fig. 4.8 presents a (partial) scheduling table produced by our mapping routine. We shall use this example to give a better intuition on the functioning of our algorithms. We assume here that the execution of task f produces data x which will be used by task g. Our scheduling table shows the result of the mapping of task g onto $Tile(2,2)$ (which also requires the mapping of the transmission of x) under the assumption that all other tasks (f, h) and data transmissions (y, z, u) were already mapped as pictured (reservations made during the mapping of g have a lighter color).

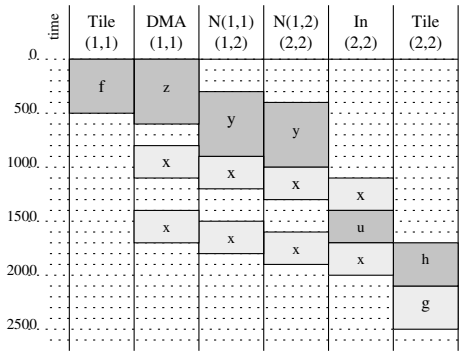

Figure 4.8: Partial scheduling table covering one communication path on our NoC. Only the 6 resources of interest are represented (out of 70). Time flows from top to bottom.

As part of the mapping of g onto $Tile(2,2)$, function ***MapCommunicationOnPath*** is called to perform the mapping of the communication of x from $Tile(1,1)$ to $Tile(2,2)$. The parameters of its call are *Path*, *StartDate*, and *DataSize*. Parameter *Path* is the list formed of resources $DMA(1,1)$, $N(1,1)(1,2)$, $N(1,2)(2,2)$, and $In(2,2)$ (the transmission route of x under the X-first routing protocol). Parameter *StartDate* is given by the end date of task f (in our case 500), and *DataSize* is the worst-case size of the data associated with the data dependency (in our case 500 32-bit words). Time is measured in clock cycles.

To minimize the overall time reserved for a data transmission, we shall require that it is never blocked waiting for a NoC resource. For instance, if the communication of x starts on the $N(1,1)(1,2)$ at date t, then on $N(1,2)(2,2)$ it must start at date $t + \underline{SegmentBufferSize}$, where $\underline{SegmentBufferSize}$ is a platform constant defining the time needed for a flit to traverse one NoC resource. In our NoC this constant is 3 clock cycles (in Fig. 4.8 we use a far larger value of 100 cycles, for clarity).

Building such synchronized reservation patterns along the communication routes is what function ***MapCommunicationOnPath*** does. It starts by obtaining the lists of free time intervals of each resource along the communication path, and realigning them by subtracting $(i-1) * \underline{SegmentBufferSize}$ from the start dates of all the free intervals of the i^{th} resource, for all i. Once this realignment is done on each resource by function ***ShiftLeftIntervals***, finding a

reservation along the communication path amounts to finding time intervals that are unused on all resources. To do this, we start by performing (in line 6 of function ***MapCommunicationOnPath***) an intersection operation returning all realigned time intervals that are free on all resources. In Fig. 4.8, this intersection operation produces (prior to the mapping of x) the intervals [800,1100) and [1400,2100]. The value 2100 corresponds here to the length of the scheduling table prior to the mapping of g.

We then call function ***ReserveIntervals*** twice, to make reservations for the data transmission and for the lock command associated with each communication. These two functions produce a list of reserved intervals, which then need to be realigned on each resource. In Fig. 4.8, these 2 calls reserve the intervals [800,1100), [1400,1700), and [1700,1704). The first 2 intervals are needed for the data transmission, and the third is used for the lock command packet.

Multiple reservations Communications are reserved at the earliest possible date, and function ***ReserveIntervals*** allows the fragmentation of a data transmission to allow a better use of NoC resources. In our example, fragmentation allows us to transmit part of x before the reservation for u. If fragmentation were not possible, the transmission of x should be started later, thus delaying the start of g, potentially lengthening the reservation table.

Procedure 2 ReserveIntervals

Input: *DataSize* : worst-case size of data to transmit
FreeIntervalList : list of free intervals before reservation
ScheduleLength : schedule length before reservation
Output: *ReservedIntervalList* : reserved intervals
NewIntervalList : list of free intervals after reservation
NewScheduleLength : schedule length after reservation

```
 1: NewIntervalList := FreeIntervalList
 2: ReservedIntervalList := ∅
 3: while DataSize > 0 and NewIntervalList≠ ∅ do
 4:    ival := GetFirstInterval(NewIntervalList);
 5:    NewIntervalList := RemoveFirstInterval(NewIntervalList);
 6:    if IntervalEnd(ival)==ScheduleLength then
 7:       RemainingIvalLength := ∞; /*ival can be extended*/
 8:    else
 9:       RemainingIvalLength := length(ival);
10:    ReservedLength := 0;
11:    while RemainingIvalLength > MinPacketSize and DataSize > 0 do
12:       /*Reserve a packet (clear, but suboptimal code)*/
13:       PacketLength := min(DataSize + PacketHeaderSize,
                              RemainingIvalLength,MaxPacketSize);
14:       RemainingIvalLength -= PacketLength;
15:       DataSize -= PacketLength - PacketHeaderSize;
16:       ReservedLength += PacketLength
17:    ReservedInterval := CreateInterval(start(ival),ReservedLength);
18:    ReservedIntervalList := AppendToList(ReservedIntervalList,ReservedInterval);
19:    if length(ival) - ReservedLength > MinPacketLength then
20:       NewIntervalList := InsertInList(NewIntervalList,
             CreateInterval(start(ival)+ReservedLength,length(ival)-ReservedLength));
21:    NewScheduleLength := max(ScheduleLength,end(ival));
```

Fragmentation is subject to restrictions arising from the way communications are packetized. An interval cannot be reserved unless it has a minimal size, allowing the transmission of at least a packet containing some payload data.

Function ***ReserveIntervals*** performs the complex translation from data sizes to needed packets and intervals reservations. We present here an unoptimized version that facilitates understanding. This version reserves one packet at a time, using a free interval as soon as it has the needed minimal size. Packets are reserved until the required *DataSize* is covered. Like for tasks, reservations are made as early as possible. For each packet reservation the cost of NoC control (under the form of the *PacketHeaderSize*) must be taken into account.

When the current scheduling table does not allow the mapping of a data communication, function ***ReserveIntervals*** will lengthen it so that mapping is possible.

Phase 3: Code generation

Once the scheduling table has been computed, executable code is automatically generated as follows: One sequential execution thread is generated for each tile and for each NoC multiplexer (resources $Tile(i,j)$, $N(i,j)(k,l)$, and $In(i,j)$ in our platform model of Section 4.5.1). The code of each thread is an infinite loop that executes the (computation or communication) operations scheduled on the associated resource in the order prescribed by their reservations. Recall that each tile contains 16 processor cores, but is reserved as a single sequential resource, parallelism being hidden inside the data-flow blocks. The sequential thread of a tile runs on the first processor core of the tile, but the code of each task can use all 16 processor cores. The code of the NoC multiplexers is executed on the router controllers.

No separate thread is generated for the DMA resource of a tile. Instead, its operations are initiated by thread of the tile. This is possible because the DMA allows the queuing of DMA commands and because mapping is performed so that activation conditions for DMA operations can be computed by the tile resource at the end of data-flow operations. For instance, in the example of Fig. 4.8, if no other operations are allocated on $Tile(0,0)$, the two DMA operations needed to send x are queued at the end of f.

The synchronization of the threads is realized by explicit lock manipulations by the processors and by the NoC control programs, which force message passing order and implicitly synchronize with the flow of passing messages. The resulting programs enforce the computed *order* of operations on each resource in the scheduling table, but allow for some timing elasticity: If the execution of an operation takes less than its WCET or WCCT, operations depending on it may start earlier. This elasticity does not compromise the worst-case timing guarantees computed by the mapping tool.

Memory handling. Our real-time scheduling and timing analysis use conservative WCET estimations for the (parallel) execution of data-flow blocks on the computing tiles, *in isolation*. Uncontrolled memory accesses coming from

other tiles during execution could introduce supplementary delays that are not (yet) taken into account in the WCET figures or by LoPhT.

To ensure the timing correctness of our real-time scheduling technique, we need to ensure that memory accesses coming from outside a tile do not interfere with memory accesses due to the execution of code inside the tile. This is done by exploiting the presence of multiple memory banks on each tile. The basic idea is to ensure that incoming DMA transfers never use the same memory banks as the code running at the same time on the CPUs. Of course, once a DMA transfer is completed, the memory banks it has modified can be used by the CPUs, the synchronization being ensured through the use of locks.

We currently ensure this property at code generation time, by explicitly allocating variables to memory banks in such a way as to exclude contentions. While not general, this technique worked well for our case studies. Integrating RAM bank allocation in the mapping algorithm is ongoing work, as is separately considering each of the computing cores of a tile (instead of considering them as a single computing resource).

4.6 Mapping (2) - Architecture-independent optimizations

The previous section showed the level of architecture detail we considered in order to allow an efficient use of computing resources. But in building the LoPhT tool, I have also taken inspiration from more general, architecture-independent optimization techniques of both classical compilation and synchronous language compilation in order to improve the quality of the generated code. These optimizations are described in references [37, 130, 121]. I will mainly discuss here the use of software pipelining techniques to improve the computation throughput of a scheduling table. In doing so I will also provide hints into a second optimization: the precise analysis of clocks to allow efficient and safe double reservation of resources in a scheduling table. the proposed algorithms are general and scalable, which allows their application to the mapping of parallelized code onto many-cores.

4.6.1 Motivation

Compilers such as GCC are expected to improve code speed by taking advantage of micro-architectural instruction level parallelism [84]. *Pipelining* compilers usually rely on *reservation tables* to represent an efficient (possibly optimal) static allocation of the computing resources (execution units and/or registers) with a timing precision equal to that of the hardware clock. Executable code is then generated that enforces this allocation, possibly with some timing flexibility. But on *VLIW architectures*, where each instruction word may start several operations, this flexibility is very limited, and generated code is virtually identical to the reservation table. The scheduling burden is mostly supported here by the compilers, which include *software pipelining* techniques [141, 2] designed

to increase the throughput of loops by allowing one loop cycle to start before the completion of the previous one.

My objective was to apply software pipelining techniques in the *system-level* off-line scheduling of embedded control specifications. The optimal scheduling of such specifications onto platforms with multiple, heterogenous execution and communication resources (distributed, parallel, multi-core) is NP-hard regardless of the optimality criterion (throughput, makespan, etc.) [67]. Existing scheduling techniques and tools ([40, 161, 76, 130, 59] or the Step 1 of Section 4.5.4) *heuristically* solve the simpler problem of synthesizing a scheduling table of *minimal length* which implements one generic cycle of the embedded control algorithm. In a hard real-time context, minimizing table length (*i.e.* the makespan) is often a good idea, because in many applications it bounds the response time after a stimulus.

But optimizing makespan alone relies on an execution model where execution cycles cannot overlap in time (no pipelining is possible), even if resource availability allows it. At the same time, most real-time applications have both makespan and throughput requirements, and in some cases achieving the required throughput is only possible if a new execution cycle is started before the previous one has completed.

This is the case in the electronic control units (ECU) of combustion engines. Starting from the acquisition of data for a cylinder in one engine cycle, an ECU must compute the ignition parameters before the ignition point of the same cylinder in the next engine cycle (a makespan constraint). It must also initiate one such computation for each cylinder in each engine cycle (a throughput constraint). On modern multiprocessor ECUs, meeting both constraints requires the use of pipelining [6]. Another example is that of systems where a faster rate of sensor data acquisition results in better precision and improved control, but optimizing this rate must not lead to the non-respect of requirements on the latency between sensor acquisition and actuation. *To allow the scheduling of such systems we consider here the static scheduling problem of optimizing both makespan and throughput, with makespan being prioritary.*

To (heuristically) solve this optimization problem, we use the three-phase implementation process defined in Section 4.5.4. The first two phases of this process, which perform allocation and scheduling and build the system-level scheduling table, are a form of *decomposed software pipelining* [152, 68, 31]. The first phase of this flow consists in applying one of the previously-mentioned makespan-optimizing tools, such as Step 1 of Section 4.5.4. The result is a scheduling table describing the execution of one generic execution cycle of the embedded control algorithm with no pipelining.

The second phase uses a novel software pipelining algorithm, introduced in this section, to significantly improve the throughput without changing the makespan and while preserving the *periodic* nature of the system. The approach has the advantage of simplicity and generality, allowing the use of existing makespan-optimizing tools.

The proposed software pipelining algorithm is a very specific and constrained form of *modulo scheduling* [140]. Like all modulo scheduling algorithms, it de-

termines a shorter *initiation interval* for the execution cycles (iterations) of the control algorithm, subject to resource and inter-cycle data dependency constraints. Unlike previous modulo scheduling algorithms, however, it starts from an already scheduled code (the non-pipelined scheduling table), and preserves all the intra-cycle scheduling decisions made at phase 1, in order to preserve the makespan unchanged. In other words, our algorithm computes the best *initiation interval* for the non-pipelined scheduling table and re-organizes resource reservations into a *pipelined scheduling table*, whose length is equal to the new initiation interval, and which accounts for the changes in memory allocation.

4.6.2 Related work and originality

Decomposed software pipelining.

Closest to our work are previous results on *decomposed software pipelining* [152, 68, 31]. In these papers, the software pipelining of a sequential loop is realized using two-phase heuristic approaches with good practical results. Two main approaches are proposed in these papers.

In the first approach, used in all 3 cited papers, the first phase consists in solving the loop scheduling problem while ignoring resource constraints. As noted in [31], existing decomposed software pipelining approaches solve this loop scheduling problem by using *retiming* algorithms. Retiming [100] can therefore be seen as a very specialized form of pipelining targeted at cyclic (synchronous) systems where each operation has its own execution unit. Retiming has significant restrictions when compared with full-fledged software pipelining:

- It is oblivious of resource allocation. As a side-effect, it cannot take into account execution conditions to improve allocation, being defined in a purely data-flow context.

- It requires that the execution cycles of the system do not overlap in time, so that one operation must be completely executed inside the cycle where it was started.

Retiming can only change the execution order of the operations inside an execution cycle. A typical retiming transformation is to move one operation from the end to the beginning of the execution cycle in order to shorten the duration (critical path) of the execution cycle, and thus improve system throughput. The transformation cannot decrease the makespan but may increase it.

Once retiming is done, the second transformation phase takes into account resource constraints. To do so, it considers the acyclic code of one generic execution cycle (after retiming). A list scheduling technique ignoring inter-cycle dependences is used to map this acyclic code (which is actually represented with a *directed acyclic graph*, or *DAG*) over the available resources.

The second technique for decomposed software pipelining, presented in [152], basically switches the two phases presented above. Resource constraints are considered here in the first phase, through the same technique used above: list

scheduling of DAGs. The DAG used as input is obtained from the cyclic loop specification by preserving only some of the data dependences. This scheduling phase decides the resource allocation and the operation order inside an execution cycle. The second phase takes into account the data dependences that were discarded in the first phase. It basically determines the fastest way a specification-level execution cycle can be executed by several successive pipelined execution cycles without changing the operation scheduling determined in phase 1 (preserving the throughput unchanged). Minimizing the makespan is important here because it results in a minimization of the memory/register use.

Originality

My student Thomas Carle defined, under my supervision, a third decomposed software pipelining technique with two significant originality points, detailed below.

Optimization of both makespan and throughput. Existing software pipelining techniques are tailored for optimizing only one real-time performance metric: the processing *throughput* of loops [158] (sometimes besides other criteria such as register usage [75, 159, 89] or code size [162]). In addition to throughput, we also seek to optimize *makespan*, with makespan being prioritary. Recall that throughput and latency (makespan) are antagonistic optimization objectives during scheduling [16], meaning that resulting schedules can be quite different.

To optimize makespan we employ in the first phase of our approach existing scheduling techniques that were specifically designed for this purpose [40, 161, 76, 130, 59]. The second phase of our flow takes the scheduling table computed in phase 1 and optimizes its throughput while keeping its makespan unchanged. This is done using a new algorithm that conserves all the allocation and intra-cycle scheduling choices made in phase 1 (thus conserving makespan guarantees), but allowing the optimization of the throughput by increasing (if possible) the frequency with which execution cycles are started.

Like retiming, this transformation is a very restricted form of modulo scheduling software pipelining. In our case, it can only change the initiation interval (changes in memory allocation and in the scheduling table are only consequences). By comparison, classical software pipelining algorithms, such as the iterative modulo scheduling of [140], perform a full mapping of the code involving both execution unit allocation and scheduling. Our choice of transformation is motivated by three factors:

- It preserves makespan guarantees.
- It gives good practical results for throughput optimization.
- It has low complexity.

The last point (complexity) is especially important when comparing our results with those of SCAN [24]. In SCAN, the objective is still the optimization of

throughput, but this objectiver is attained through an exploration along two dimensions: throughput and time horizon (a notion closely related to makespan). Through this bi-criteria exploration, SCAN is similar to our work. However, optimization does not establish the priority of time horizon over throughput and, even more important, exploration steps of SCAN rely on exact solving of linear programming problems, which is largely more complex than the algorithms we propose [72]. Addressing this complexity is actually the main objective of SCAN, and the main tuning parameter of the algorithm is the timeout value at which exploration of a particular point in the search space is abandoned.

It is important to note that our transformation is not a form of retiming. Indeed, it allows for a given operation to span over several cycles of the pipelined implementation, and it can take advantage of conditional execution to improve pipelining, whereas retiming techniques work in a pure data-flow context, without predication.

Predication. Embedded control specifications often include conditional control, for instance under the form of execution modes. For an efficient mapping of such specifications, it is important to allow an independent, predicated (conditional) control of the various computing resources. However, most existing techniques for software pipelining [2, 153, 158] use execution platform models that significantly constrain or simply prohibit predicated resource control. This is due to limitations in the hardware targeted by classical software pipelining (processor pipelines). One common problem is that two different operations cannot be scheduled at the same date on a given resource (functional unit), even if they have exclusive predicates (like the branches of a test). The only exception we know to this rule is *predicate-aware scheduling (PAS)* [146].

By comparison, the computing resources of our target architectures are not mere functional units of a CPU (as in classical predicated pipelining), but full-fledged processors such as PowerPC, ARM, *etc.* The operations executed by these computing resources are large sequential functions, and not simple CPU instructions. Thus, each computing resource allows unrestricted predication control by means of conditional instructions, and the timing overhead of predicated control is negligible with respect to the duration of the operations. This means that our architectures satisfy the PAS requirements. The drawback of PAS is that sharing the same resource at the same date is only possible for operations of the same cycle, due to limitations in the dependency analysis phase. Our technique removes this limitation.

To exploit the full predicated control of our platform we rely on a new intermediate representation, namely *predicated and pipelined scheduling tables*. By comparison to the modulo reservation tables of [98, 140], our scheduling tables allow the explicit representation of the execution conditions (predicates) of the operations. In turn, this allows the double reservation of a given resource by two operations with exclusive predicates.

Other work on pipelining for task scheduling

A significant amount of work exists on applying software pipelining or retiming techniques for the efficient scheduling of tasks onto coarser-grain architectures, such as multi-processors [95, 156, 45, 48, 40, 111]. To our best knowledge, these results share the two fundamental limitations of other software pipelining algorithms: Optimizing for only one real-time metric (throughput) and not fully taking advantage of conditional execution to allow double allocation of resources.

4.7 Conclusion

The most important conclusion of the work presented here is that combining techniques of both compilation and real-time scheduling is possible. Taking into account precise architectural detail and using state-of-the-art optimizations actually gave results beyond our expectations. For very regular applications such as the previously-mentioned platooning application and the FFT, LoPhT was able to generate code whose observed latency and throughput is within 1% of the predicted values. Furthermore, the code produced by LoPhT for the FFT ran faster than a classical NoC-based parallel implementation of the FFT [13] running on our architecture. In other words, **our tool produced code that not only has statically-computed hard real-time bounds (which the hand-written code has not) but is also faster**.

Of course, part of these results is due to the regularity of the applications and to the hardware architecture, which has very good support for off-line real-time scheduling. But the results are also due to the careful definition of the software architecture, and to the definition of the scheduling and code generation algorithms, which take into account fine detail of the hardware while using efficient general-purpose optimizations.

And the good news is that much more remains to be gained. With my collaborators, I am currently investigating the mapping of more dynamic applications (*e.g.* h264/hevc video codecs), the definition of mapping techniques where processor cores and memory banks are individually considered by the mapping algorithms, the application of other classical compiler optimizations, the definition of general architecture description languages, *etc.*

Chapter 5

Automatic implementation of systems with complex functional and non-functional properties

In the previous chapter I explained how a classical compilation approach was adapted to allow precise and conservative timing accounting, and thus provide hard real-time throughput and latency guarantees for code executed on complex platforms (many-cores). The approach combines general-purpose and architecture-specific optimizations to produce very efficient code. Such a compilation approach never fails if execution on the platform is functionally possible, because no real-time requirements are taken into account.

But the implementation needs of complex embedded systems are not well captured under the form of such unconstrained optimization problems. Complex embedded systems have multiple *non-functional requirements* that must be satisfied by the final implementation, and which need to be taken into account by the scheduling and code generation algorithms. From this perspective, the work presented in the previous chapter, for all its good results, is only an enabler technology in the definition of a system-level compilation approach for complex embedded systems.

In this chapter I explain how, together with my students and post-docs, and starting from the results of the previous chapter, I extended the LoPhT tool to take into account multiple non-functional requirements, thus completing its design as a real-time systems compiler.

LoPhT considers all the modeling and code generation needs of a certain class of embedded control systems, characterized by the use of off-line real-time scheduling and by the use of a time-triggered interface with the physical environment. This class of systems includes systems requiring space-time isolation,

as mandated by the IMA/ARINC 653 standard [8]. The design of LoPhT has been driven by industrial case studies requiring space-time isolation and coming from the aerospace and rail industries [38, 49]. To cover the needs of these systems, LoPhT considers the following non-functional properties:

- Preemptability[1] of the operations of the application. We do not consider here the full force of preemptive execution, like in Giotto [85] or Prelude [116]. Instead, we consider a more temporally predictable approach, and rely on *pre-computed preemption*, where all possible preemption dates are statically computed at off-line scheduling time.

- Space-time partitioning of the application, as specified by the ARINC653 [8] standard. Time partitioning is also that of TTA [97] and FlexRay [142] (the static segment), allowing the static allocation of CPU or bus time slots, on a periodic basis, to various parts (known as partitions) of the application. Also known as static time division multiplexing (TDM) scheduling, time partitioning further enhances the temporal determinism of a system.

- Release dates and deadlines for the tasks of the application, which allow the modeling of constraints coming from the environment and/or the dynamics of the system. An important point here is that we allow the use of deadlines that are longer than the periods in the specification. This allows a more natural real-time specification, improved schedulability, and less context changes between partitions (which are notoriously expensive).

All these come in addition to allocation constraints, which were already taken into account in the previous chapter.

Work on integrating these non-functional requirements naturally raised some optimization and code generation questions that were addressed by our work on LoPhT: the minimization of the preemptions, the minimization of the number of tasks, and the synthesis of inter-partition communication code.

5.1 Related work

The main originality of this work was to define a complex task model allowing the specification of all the functional and non-functional aspects needed for the corect and efficient implementation of our systems. Of course, *prior work already considers all these functional and non-functional aspects, but either in isolation (one aspect at a time), or through combinations that do not cover the modeling needs of the target systems.* Our contributions are the non-trivial combination of these aspects in a coherent formal model and the definition of synthesis algorithms able to build a running real-time implementation.

[1] The use of the terms preemptability and preemptable is common in real-time when referring to tasks. The terms “interrupt” and “interruptible” refer to the machine interrupts instead.

Previous work [86, 85, 116, 106, 3] on the implementation of multi-periodic synchronous programs and the work by [25] and [47] on the scheduling of dependent task systems have been important sources of inspiration. By comparison, our work provides a general treatment of ARINC 653-like partitioning and of conditional execution, and a novel use of deadlines longer than periods to allow faithful real-time specification.

The work of [40] addresses the multiprocessor scheduling of synchronous programs under bus partitioning constraints. By comparison, our work takes into account conditional execution and execution modes, allows preemptive scheduling, and allows automatic allocation of computations and communications. Taking advantage of the time-triggered execution context, our approach also relies on fixed deadlines (as opposed to relative ones), which facilitates the definition of fast mapping heuristics.

Another line of work on the scheduling of dependent tasks is represented by the works of [119] and [161]. In both cases, the input of the technique is a DAG, whereas our functional specifications allow the use of delayed dependencies between successive iterations of the DAG. Other differences are that the technique [161] does not take into account ARINC 653-like partitioning or conditional execution, and the technique of [119] does not allow the specification of complex end-to-end latency constraints. [65] does consider conditional control, but does so in a mono-processor, non-partitioned, non-preemptive context.

The off-line (pipelined) scheduling of tasks with deadlines longer than the periods has been previously considered (among others) by [66], but this work does not consider, as we do, partitioning constraints and the use of execution conditions to improve resource allocation. This is also our originality with respect to other classical work on static scheduling of periodic systems [139].

Compared to previous work by [66] on real-time scheduling for predictable, yet flexible real-time systems, our approach does not directly cover the issue of sporadic tasks, but allows a more flexible treatment of periodic (time-triggered) tasks. Based on a different representation of real-time characteristics and on a very general handling of execution conditions, we allow for better flexibility *inside* the fully predictable domain.

From an implementation-oriented perspective, Giotto [85, 86], ΨC [103], and Prelude [116, 136] make the choice of mixing a globally time-triggered execution mechanism with on-line priority-driven scheduling algorithms such as RM or EDF. By comparison, we made the choice of taking *all* scheduling decisions off-line. Doing this complicates the implementation process, but imposes a form of temporal isolation between the tasks which reduces the number of possible execution traces and increases timing precision (as the scheduling of one task no longer depends on the run-time duration of the others). In turn, this facilitates verification and validation. Furthermore, a fully off-line scheduling approach such as ours has the potential of improving worst-case performance guarantees by taking better decisions than a RM/EDF scheduler which follows an as-soon-as-possible (ASAP) scheduling paradigm. For instance, reducing the number of notoriously expensive partition changes (detailed in [38]) uses a scheduling technique that is not ASAP. These partition changes are not taken into account

in the optimality results concerning the EDF scheduling of Prelude [116].

Compared to classical work on the on-line real-time scheduling of tasks with execution modes (*cf.* [14]), our off-line scheduling approach comes with precise control of timing, causalities, and the correlation (exclusion relations) between multiple test conditions of an application. It is therefore more interesting for us to use a task model that explicitly represents execution conditions. We can then use *table-based scheduling* algorithms that precisely determine when the same resource can be allocated at the same time to two tasks because they are never both executed in a given execution cycle, as expained in the previous chapter.

The use of execution conditions to allow efficient resource allocation is also the main difference between our work and the classical results of [155]. Indeed, the exclusion relation defined by Xu does not model conditional execution, but resource access conflicts, thus being fundamentally different from the exclusion relation we defined in Section 4.5.2. Our technique also allows the use of execution platforms with non-negligible communication costs and multiple processor types, as well as the use of preemptive tasks (unlike in Xu's paper).

The off-line scheduling on partitioned ARINC 653 platforms has been previously considered, for instance by Al Sheikh *et al.* [143] and by Brocal *et al.* in Xoncrete [29]. The first approach only considers systems with one task per partition, whereas our work considers the general case of multiple tasks per partition. The second approach (Xoncrete) allows multiple tasks per partition, but does not seem interested in having a functionally deterministic specification and preserving its semantics during scheduling (as we do). For instance, its input formalism specifies not periods, but ranges of acceptable periods, and the first implementation step adjusts these periods to reduce their lowest common multiple (thus changing the semantics). Other differences are that our approach can take into account conditional execution and execution modes, and that we allow scheduling onto multi-processors, whereas Xoncrete does not.

More generally, our work is related to work on the scheduling for precision-timed architectures (*e.g.* [57]). Our originality is to consider complex non-functional constraints. The work on the PharOS technology [103] also targets dependable time-triggered system implementation, but with two main differences. First, we follow a classical ARINC 653-like approach to temporal partitioning. Second, we take all scheduling decisions off-line. This constrains the system but reduces the scheduling effort needed from the OS, and improves predictability.

5.2 Time-triggered systems

5.2.1 General definition

By *time-triggered systems* we understand systems satisfying the following 3 properties:

TT1 A system-wide time reference exists, with good-enough precision and ac-

curacy. We shall refer to this time reference as the *global clock*.[2] All timers in the system use the global clock as a time base.

TT2 The execution duration of code driven by interrupts other than the timers (*e.g.* interrupt-driven driver code) is negligible. In other words, for timing analysis purposes, code execution is only triggered by timers synchronized on the global clock.

TT3 System inputs are only read/sampled at timer triggering points.

This definition places no constraints on the sequential code triggered by timers. In particular:

- Classical sequential control flow structures such as *sequence* or *conditional execution* are permitted, allowing the representation of modes and mode changes.
- Timers are allowed to preempt the execution of previously-started code.

This definition of time-triggered systems is fairly general. It covers single-processor systems that can be represented with *time-triggered e-code programs*, as they are defined by Henzinger and Kirsch [86]. It also covers multiprocessor extensions of this model, as defined by Fischmeister et al. [64] and used in [124]. In particular, our model covers time-triggered communication infrastructures such as TTA and FlexRay (static and dynamic segments) [97, 142], the periodic schedule tables of AUTOSAR OS [11], as well as systems following a multi-processor periodic scheduling model without jitter and drift.[3] It also covers the execution mechanisms of the avionics ARINC 653 standard [8] provided that interrupt-driven data acquisitions, which are confined to the ARINC 653 kernel, are presented to the application software in a time-triggered fashion satisfying property *TT3*. One way of ensuring that *TT3* holds is presented in [107], and to our knowledge, this constraint is satisfied in all industrial settings.

5.2.2 Model restriction

The major advantage of time-triggered systems, as defined above, is that they have the property of *repeatable timing* [58]. Repeatable timing means that for any two input sequences that are identical in the large-grain timing scale determined by the timers of a program, the behaviors of the program, including timing aspects, are identical. This property is extremely valuable in practice because it largely simplifies debugging and testing of real-time programs. A time-triggered platform also insulates the developer from most problems stemming from interrupt asynchrony and low-level timing aspects.

However, the applications we consider have even stronger timing requirements, and must satisfy a property known as *timing predictability* [58]. Timing

[2] For single-processor systems the global clock can be the CPU clock itself. For distributed multiprocessor systems, we assume it is provided by a platform such as TTA [97] or by a clock synchronization technique such as the one of Potop *et al.* [124].

[3] But these two notions must be accounted for in the construction of the global clock [124].

predictability means that formal timing guarantees covering *all* possible executions of the system should be computed off-line by means of (static) analysis. The general time-triggered model defined above remains too complex to allow the analysis of real-life systems. To facilitate analysis, this model is usually restricted and used in conjunction with WCET analysis of the sequential code fragments.

In my work I considered a commonly-used restriction of the general definition provided above. In this restriction, timers are triggered following a fixed pattern which is repeated periodically in time. Following the convention of ARINC 653, we call this period the *major time frame (MTF)*. The timer triggering pattern is provided under the form of a set of fixed offsets $0 \leq t_1 < t_2 < \ldots < t_m < MTF$ defined with respect to the start of each *MTF* period. Note that the code triggered at each offset may still involve complex control, such as conditional execution or preemption.

This restriction corresponds to the classical definition of time-triggered systems by Kopetz [96, 97]. It covers our target platform, TTA, FlexRay (the static segment), and AUTOSAR OS (the periodic schedule tables). At the same time, it does not fully cover ARINC 653. As defined by this standard, partition scheduling is time-triggered in the sense of Kopetz. However, the scheduling of tasks inside partitions is not, because periodic processes can be started (in normal mode) with a release date equal to the current time (not a predefined date). To fit inside Kopetz's model, an ARINC 653 system should not allow the start of periodic processes *after* system initialization, *i.e.* in normal mode.

5.2.3 Temporal partitioning

Our target architectures follow the strong temporal partitioning paradigm of ARINC 653. In this paradigm, both system software and platform resources are *statically* divided among a finite set of *partitions* $Part = \{part_1, \ldots, part_k\}$. Intuitively, a partition comprises both a software application of the system and the execution and communication resources allocated to it. The aim of this static partitioning is to limit the functional and temporal influence of one partition on another. Partitions can communicate and synchronize only through a set of explictly-specified inter-partition channels.

To eliminate timing interference between partitions running on a processor, the static partitioning of the processor time is done using a static time division multiplexing (TDM) mechanism. In our case, the static TDM mechanism is built on top of the time-triggered model of the previous section. It is implemented by partitioning, separately for each processor P_i, the *MTF* defined above into a finite set of non-overlapping *windows* $W_i = \{w_i^1, \ldots, w_i^{k_i}\}$. Each window w_i^j has a fixed start offset tw_i^j, a duration dw_i^j, and it is either allocated to a single partition $partw_i^j$, or left unused.

Software belonging to partition $part_i$ can only be executed during windows belonging to $part_i$. Unfinished partition code will be preempted at window end, to be resumed at the next window of the partition. There is an implicit

assumption that the scheduling of operations inside the *MTF* will ensure that non-preemptive operations will not cross window end points. For our scheduling algorithms, the partitioning of the *MTF* into windows can be either an input or an output. More precisely, all, none, or part of the windows can be provided as input.

5.3 A typical case study.

A typical case study that can be fully handled by LoPhT is a launcher (spacecraft) embedded control system model provided by Airbus DS (formerly ASTRIUM). Such spacecraft systems have very strict real-time requirements, as the unavailability of the avionics system of a space launcher during a few milliseconds in the atmospheric phase may lead to the destruction of the launcher. In a launcher control system, latency real-time requirements are defined between the acquisition of data by sensors and the sending of orders to actuators. These requirements apply to the control algorithms (GNC, for Guidance, Navigation and Control algorithms), which are usually implemented on a dedicated processor in a classical multi-tasking approach. In the last decade, the increase in raw computational power of space processors allowed the distribution of the GNC computations on the processors of the sensors and actuators, and the suppression of the processor that was, until now, dedicated to the GNC computations. In the future, the GNC algorithms could be separated, and for example the navigation algorithm could run on the processor controlling the gyroscope, while the control algorithm would run on the processor controlling the thruster. For the companies who manufacture the spacecraft systems (space launchers or space transportation vehicles), this means a non-negligeable reduction in weight and power consumption, which ultimately amounts to a reduction in costs.

But distributing GNC code onto sensor and actuator processors leads to situations where a processor is shared by pieces of software having different Design Assurance Levels (such as gyroscope control and navigation), and consequently requires the use of an operating system that enforces Time and Space Partitioning (TSP) between them. In such operating systems, scheduling is handled by a hierarchic two-level scheduler in which the top level is of static Time-triggered (TDM) type. This is the case, for instance, in ARINC 653-compliant systems [8]. In some systems, such as future launchers, predictability concerns go even farther, and the processors of the distributed implementation platform share a common time base, allowing a globally time-triggered implementation. This is the type of system we consider: distributed systems that are time triggered at all scheduling levels. Such systems offer the best predictability and allow the computation of very tight worst-case response time bounds.

In the context of this case study, the LoPhT tool solves the following real-time implementation problem: Given a set of end-to-end latencies defined at spacecraft system level, along with sensing and actuation operations offsets and safe worst case execution time (WCET) estimations of the computation functions, synthesize the time-triggered schedule of the system, including the activa-

tion of each partition and each functional node, and the bus frame. Scheduling must guarantee the respect of all non-functional requirements, and be accompanied by the generation of all implementation files (executable code and system configuration files).

5.3.1 Functional specification

The first step in handling this case study was to derive a task model in our formalism. We use here its simplified version, the full example being presented in [38].

During modeling, we discovered that the initial system was over-specified, in the sense that real-time constraints were imposed in order to ensure causal ordering of tasks instances using AADL constructs. Removing these constraints and replacing them with (less constraining) data dependencies gave us more freedom for scheduling, allowing for a reduction in the number of partition changes. The resulting specification is presented in Fig. 5.1.

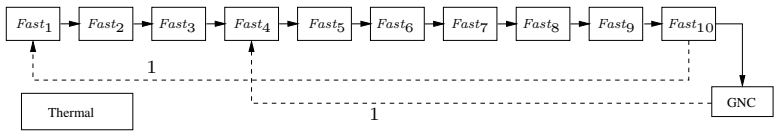

Figure 5.1: The *GNCSimple* example

Our model, named *GNCSimple* represents a system with 3 tasks *Fast*, *GNC*, and *Thermal*. The periods of the 3 tasks are 10ms, 100ms, and 100ms, respectively, meaning that *Fast* is executed 10 times for each execution of *GNC* and *Thermal*. The hyperperiod expansion described in [38] produces a single-clock synchronous program represented by the dependent task system of Fig. 5.1. Recall from Section 4.5.2 that the solid arcs connecting the tasks $Fast_i$ and *GNC* represent regular (intra-cycle) data dependencies. Delayed data dependencies of depth 1 represent the transmission of information from one MTF to the next. In this simple model, task *Thermal* has no dependencies.

5.4 Non-functional properties

Our task model considers non-functional properties of 4 types: real-time, allocation, partitioning, and preemptability.

5.4.1 Period, release dates, and deadlines

As explained in Chapter 2, the initial functional specification of a system is usually provided by the control engineers, which must also provide a platform-independent real-time specification in terms of *periods*, *release dates*, and *deadlines*. This specification is directly derived from the analysis of the control sys-

tem, and does not depend on architecture details such as number of processors, speed, *etc.* The architecture may impose its own set of real-time characteristics and requirements. Our model allows the specification of all these characteristics and requirements in a specific form adapted to our functional specification model and time-triggered implementation paradigm.

Period. Recall from the previous section that after hyper-period expansion all the tasks of a dependent task system D have same period. We shall call this period the *major time frame* of the dependent task system D and denote it $MTF(D)$. We will require it to be equal to the MTF of its time-triggered implementation, as defined in Section 5.2.2.

Throughout this chapter, we will assume that $MTF(D)$ is an input to our scheduling problem. Other scheduling heuristics, such as those of Chapter 4 can be used in the case where the MTF must be computed.

Release dates and deadlines. For each task $\tau \in T(D)$, we allow the definition of a release date $r(\tau)$ and a deadline $d(\tau)$. Both are positive offsets defined with respect to the start date of the current MTF (period). To signify that a task has no release date constraint, we set $r(\tau) = 0$. To signify that it has no deadline we set $d(\tau) = \infty$.

The main intended use of release dates is to represent constraints related to input acquisition. Recall that in a time-triggered system all inputs are sampled. We assume in our work that these sampling dates are known (a characteristic of the execution platform), and that they are an input to our scheduling problem. This is why they can be represented with fixed time offsets. Under these assumptions, a task using some input should have a release date equal to (or greater than) the date at which the corresponding input is sampled.

End-to-end latency requirements are specified using a combination of both release dates and deadlines. We require that end-to-end latencies are defined on flows (chains of dependent task instances) starting with an input acquisition and ending with an output. Since acquisitions have fixed offsets represented with the release dates, the latency constraints can also be specified using fixed offsets, namely the deadlines.

Before providing an example, it is important to recall that our real-time implementation approach is based on off-line scheduling. The release dates and deadlines defined here are specification objects used by the off-line scheduler alone. These values have no direct influence on implementations, which are exclusively based on the scheduling table produced off-line. In the implementation, release dates are always equal to the start dates computed off-line, which can be very different from the specification-level release dates.

5.4.2 Modeling of the case study

The specification in Fig. 5.2 adds a real-time characterization to the *GNCSimple* example of Fig. 5.1. Here, $MTF(GNCSimple) = 100$ ms. Release dates and

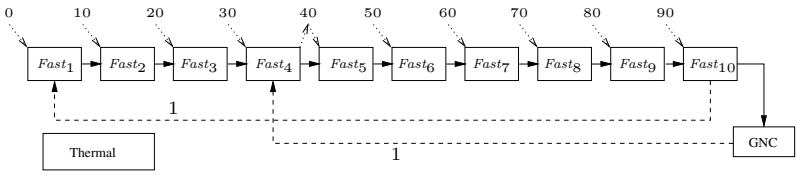

Figure 5.2: Real-time characterization of the *GNCSimple* example (MTF = 100 ms)

deadlines are respectively represented with descending and mounting dotted arcs. The release dates specify that task *Fast* uses an input that is sampled with a period of 10ms, starting at date 0, which imposes a release date of $(n-1)*10$ for $Fast_n$. Note that the release dates on $Fast_n$ constrain the start of *GNC*, because *GNC* can only start after $Fast_{10}$. However, we do not consider these constraints to be a part of the specification itself. Thus, we set the release dates of tasks *GNC* and *Thermal* to 0 and do not represent them graphically.

Only task $Fast_4$ has a deadline that is different from the default ∞. In conjunction with the 0 release date on $Fast_1$, this deadline represents an end-to-end constraint of 140ms on the *flow* defined by the chain of dependent task instances

$$Fast_1{}^n \rightarrow Fast_2{}^n \rightarrow \ldots \rightarrow Fast_{10}{}^n \rightarrow GNC^n \rightarrow Fast_4{}^{n+1}$$

for $n \geq 0$. Formally, it requires that no more that 140ms separate the start of the n^{th} instance of task $Fast_1$ from the end of the $(n+1)^{th}$ instance of task $Fast_4$. Since the release date of task instance $Fast_1{}^n$ in the *MTF* of index n is 0, this flow constraint translates into the requirement that $Fast_4{}^{n+1}$ terminates 140ms after the beginning of the *MTF* of index n. This is the same as 40ms after the beginning of *MTF* of index $n+1$ (because the length of one *MTF* is 100ms). The deadline of $Fast_4$ is therefore set to 40ms.

5.4.3 Architecture-dependent constraints

The period, release dates and deadlines of Fig. 5.2 represent architecture-independent real-time requirements that must be provided by the control engineer. But architecture details may impose constraints of their own. For instance, assume that the samples used by task *Fast* are stored in a 3-place circular buffer. At each given time, *Fast* uses one place for input, while the hardware uses another to store the next sample. Then, to avoid buffer overrun, the computation of $Fast_n$ must be completed before date $(n+1)*10$, as required by the new deadlines of Fig. 5.3. Note that these deadlines can be both larger than the period of task *Fast*, and larger than the *MTF* (for $Fast_{10}$). By comparison, the specification of Fig. 5.2 corresponds to the assumption that input buffers are infinite, so that the architecture imposes no deadline constraint. Also note in Fig. 5.3 that the deadline constraint on $Fast_3$ is redundant, given the deadline of $Fast_4$

and the data dependency between $Fast_3$ and $Fast_4$. Such situations can easily arise when constraints from multiple sources are put together, and do not affect the correctness of the scheduling approach.

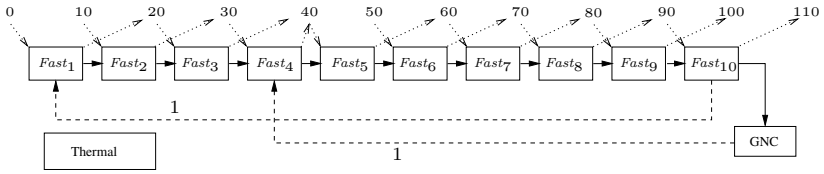

Figure 5.3: Adding 3-place circular buffer constraints to our example

5.4.4 Worst-case durations, allocations, preemptability

We also need to describe the processing capabilities of the various processors and the bus:

For each data type t associated with a dependency (simple or delayed), we define the worst-case memory footprint of a value of type t. This information allows the computation of the *worst-case communication time (WCCT)* for a data of that type, using the formula of Section 4.5.1.

- For each task $\tau \in T(D)$ and each processor $P \in Procs(Arch)$ we provide the *capacity*, or *duration* of τ on P. We assume this value is obtained through a worst-case execution time (WCET) analysis, and denote it $WCET(\tau, P)$.[4] This value is set to ∞ when execution of τ on P is not possible.
- Like in Section 4.5.3, for each data type $type(a)$ used in the specification, we provide $WCCT(type(a))$ as an upper bound on the transmission time of a value of type $type(a)$ over the bus.[5] We assume this value is always finite.

Note that the $WCET$ information may implicitly define *absolute allocation constraints*, as $WCET(t, P) = \infty$ prevents t from being allocated on P. Such allocation constraints are meant to represent hardware platform constraints, such as the positioning of sensors and actuators, or designer-imposed placement constraints. *Relative allocation constraints* can also be defined, under the form of *task groups* which are subsets of $T(D)$. The tasks of a task group must be allocated on the same processor.

Our task model allows the representation of both preemptive and non-preemptive tasks. The preemptability information is represented for each task τ by the flag $is\_preemptive(\tau)$.

[4]Note the difference with respect to the notations of Section 4.5.3, justified by the fact that multiprocessor architectures are here heterogenous, as opposed to homogenous for the MPPAs considered in the previous chapter.

[5]We make the simplifying assumption that the architecture uses a single broadcast bus.

5.4.5 Partitioning

Recall from Section 5.2.3 that there are two aspects to partitioning: the partitioning of the application and that of the resources (in our case, CPU time). On the application part, we assume that every task τ belongs to a partition $part_\tau$ of a fixed partition set $Part = \{part_1, \ldots, part_k\}$.

Also recall from Section 5.2.3 that CPU time partitioning, *i.e* the time windows on processors and their allocation to partitions can be either provided as part of the specification or computed by our algorithms. Thus, our specification may include window definitions which cover none, part, or all of CPU time of the processors. LoPhT does not currently allow the specification of a partitioning of the shared bus (this is ongoing work described in [73]).

5.5 Scheduling and code generation

The definition of our task model is now completed. It allows the specification of all the functional and non-functional aspects needed for the corect and efficient implementation of our target class of systems. It also allows the application of algorithms allowing the fully automatic synthesis of implementations. By implementations we mean here the full code of tasks, plus the configuration of the ARINC 653 OSs running on processors, and the schedule of the bus.

The mapping technique is developed on top of the principles and algorithms described in the previous chapter. One important point is that proving the correctness of our algorithms requires significant investment in proving that the platform model used by the scheduling algorithms is a conservative abstraction of the actual execution platform including hardware, ARINC 653 operating system, and drivers. For instance, we assume in our work that the impact of the OS/scheduler and driver (I/O) code on the WCETs can be bounded.

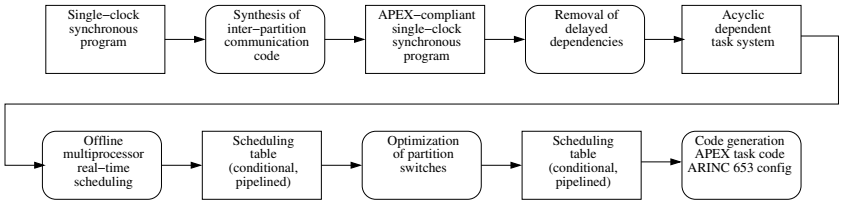

Figure 5.4: LoPhT transformation flow

I will not provide here details of individual scheduling and code generation algorithms (more information can be found in [38, 34]). Instead, I will focus on presenting the global flow of transformations, which is more complex than than of the previous chapter. As Fig. 5.4 shows, it starts from a single-clock synchronous program. Non-functional requirements specify the partition of each data-flow block. Based on them, the initial program is transformed to ensure that every communication that crosses the boundary of a partition is performed

using the operations prescribed by the APEX API of ARINC 653 [8]. This transformation can be quite complex in the presence of conditional computations and communications.

The second step transforms the APEX-compliant synchronous program into a into an acyclic dependent task system. Doing this will allow in the next section the use of simpler scheduling algorithms that work on acyclic task graphs. The main element of complexity of this transformation is the replacement of delayed dependencies (which may cause cycles) with real-time constraints. Clearly, doing this may introduce real-time requirements (deadlines) that were not part of the original specification, which in turn implies that the method is non-optimal (it is a heuristic).

The next step performs off-line real-time multiprocessor scheduling. The algorithm we designed for this purpose includes the platform-independent optimizations introduced in the previous chapter: software pipelining and safe double reservation. However, scheduling is overall driven by a deadline-driven routine specifically designed to take into account our non-functional requirements: real-time, partitioning, preemptability and allocation. The result is a scheduling heuristic of low-complexity (which ensures scalability) but which gives good practical results. Scalability is an important factor in the design of our heuristics, because the complexity of applications in both hardware and software is rapidly increasing. For instance, taking into account TTEthernet-based architectures (work in progress [73])requires taking into account complex network descriptions, similar to those of the networks-on-chips of the previous chapter.

The scheduling algorithm follows a classical ASAP (as-soon-as-possible) deadline-driven scheduling policy, which is good for ensuring schedulability. However, resulting schedules may have a lot of unneeded preemptions and, most importantly, partition switches which are notoriously expensive. To reduce the number of partition switches, we perform a heuristic post-scheduling optimization of our scheduling tables. The optimized scheduling tables are then used to generate implementation code and system configuration files.

5.6 Conclusion

The conclusion is brief: I believe that my work of the past years (presented in this thesis) shows that *real-time systems compilation* is an attainable goal for industrially-significant classes of embedded systems. Of course, my work, and that of my collaborators, is just a first step in this direction, and a lot of work remains to be done. There are scientific and technical challenges, many of them already mentioned in this thesis, or in the papers I published. Among these, I will only mention here one: moving to larger, more dynamic classes of specifications and implementations, but without losing too much of the predictability and/or efficiency. However, the most important challenge of all is not technical, nor scientific. It is that of ensuring community and industrial acceptance. I can only hope is that my work will pass, in time, this test.

Bibliography

[1] The Epiphany many-core architecture. `www.adapteva.com`, 2012.

[2] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3), September 1995.

[3] M. Alras, P. Caspi, A. Girault, and P. Raymond. Model-based design of embedded control systems by means of a synchronous intermediate model. In *Proceedings ICESS*, pages 3–10, Zhejiang, China, May 2009.

[4] P. Amagbégnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Proceedings PLDI'95*, La Jolla, CA, USA, June 1995.

[5] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R.M. Rabbah, and W. Thies. Language and compiler design for streaming applications. *Int. J. Parallel Program.*, 33(2), June 2005.

[6] C. André, F. Mallet, and M.-A. Peraldi-Frati. A multiform time approach to real-time system modeling; application to an automotive system. In *Proceedings SIES*, Lisbon, Portugal, July 2007.

[7] Charles André. Computing syncharts reactions. *Electron. Notes Theor. Comput. Sci.*, 88, October 2004.

[8] ARINC 653: Avionics application software standard interface. `www.arinc.org`, 2005.

[9] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. Dupont de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. Morey Chaisemartin, Thanh Hai Nguyen, X. Raynaud, and R. Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In *Proceedings ALCHEMY 2013*, Barcelona, Spain, June 2013.

[10] C. Auger. *Compilation certifiée de SCADE/LUSTRE*. PhD thesis, Universit Paris Sud, 2013. In French.

[11] Autosar (automotive open system architecture), release 4. `http://www.autosar.org/`, 2009.

[12] Y. Aydi, M. Baklouti, M. Abid, and J.-L. Dekeyser. A multi-level design methodology of multistage interconnection network for mpsocs. *IJCAT*, 42(2/3):191–203, 2011.

[13] Jun Ho Bahn, Jungsook Yang, and N. Bagherzadeh. Parallel FFT algorithms on network-on-chips. In *Proceedings ITNG 2008*, april 2008.

[14] Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.

[15] H. Bekker and E.J. Dijkstra. Delay-insensitive synchronization on a message passing architecture with an open collector bus. In *Parallel and Distributed Processing, 1996. PDP '96. Proceedings of the Fourth Euromicro Workshop on*, pages 75–79, Jan 1996.

[16] A. Benoît, V. Rehn-Sonigo, and Y. Robert. Multi-criteria scheduling of pipeline workflows. In *Proceedings of the International Conference on Cluster Computing*, Austin, TX, USA, Sep 2007.

[17] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125 – 171, 2000.

[18] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[19] A. Benveniste and P. Le Guernic. Hybrid dynamical systems and the signal programming language. *IEEE Trans. Automat. Control*, 35:535–546, May 1990.

[20] J.L. Bergerand, P. Caspi, D. Pilaud, N. Halbwachs, and E. Pilaud. Outline of a real time data flow language. In *Proceedings RTSS*, San Diego, CA, USA, December 1985.

[21] G. Berry, S. Moisan, and J.-P. Rigault. Esterel: Towards a synchronous and semantically sound high-level language for real-time applications. In *Proceedings RTSS*, Arlington, VA, USA, 1983. IEEE Catalog 83CH1941-4.

[22] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. *IEEE Transactions on Signal Processing*, 44:397–408, Feb. 1996.

[23] T. Bjerregaard and J. Sparso. Implementation of guaranteed services in the mango clockless network-on-chip. *Computers and Digital Techniques*, 153(4), 2006.

[24] F. Blachot, Benoit Dupont de Dinechin, and Guillaume Huard. Scan: A heuristic for near-optimal software pipelining. In WolfgangE. Nagel, WolfgangV. Walter, and Wolfgang Lehner, editors, *Euro-Par 2006 Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science*, pages 289–298. Springer Berlin Heidelberg, 2006.

[25] J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proceedings of the International Workshop Organized by the Commision of the European Communities on Modelling and Performance Evaluation of Computer Systems*, pages 57–65, Amsterdam, The Netherlands, The Netherlands, 1977. North-Holland Publishing Co.

[26] S. Borkar. Thousand core chips – a technology perspective. In *Proceedings DAC*, San Diego, CA, USA, 2007.

[27] O. Bouissou and A. Chapoutot. An operational semantics for simulink's simulation engine. In *Proceedings LCTES*, pages 129–138, 2012.

[28] T. Bourke and M. Pouzet. Zelus: A synchronous language with ODEs. In *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, pages 113–118, Philadelphia, USA, March 2013.

[29] V. Brocal, M. Masmano, I. Ripoll, A. Crespo, and P. Balbastre. Xoncrete: a scheduling tool for partitioned real-time systems. In *Proceedings ERTS*, Toulouse, France, 2010.

[30] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *International Journal in Computer Simulation*, 4(2), 1994.

[31] P.-Y. Calland, A. Darte, and Y. Robert. Circuit retiming applied to decomposed software pipelining. *Parallel and Distributed Systems, IEEE Transactions on*, 9(1):24–35, 1998.

[32] S.L. Campbell, J.-P. Chancelier, and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer, 2010. Second edition.

[33] E. Carara, N. Calazans, and F. Moraes. Router architecture for high-performance nocs. In *Proceedings SBCCI*, Rio de Janeio, Brazil, 2007.

[34] T. Carle. *Efficient compilation of embedded control specifications with complex functional and non-functional properties*. PhD thesis, EDITE, Paris, France, 2014.

[35] Thomas Carle, Manel Djemal, Daniela Genius, François Pêcheux, Dumitru Potop-Butucaru, Robert de Simone, Franck Wajsbürt, and Zhen Zhang. Reconciling performance and predictability on a many-core through off-line mapping. In *Proceedings of the 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip, ReCoSoC 2014, Montpellier, France, May 26-28, 2014*, 2014.

[36] Thomas Carle, Manel Djemal, Dumitru Potop-Butucaru, Robert de Simone, and Zhen Zhang. Static mapping of real-time applications onto massively parallel processor arrays. In *Proceedings of the 14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*, 2014.

[37] Thomas Carle and Dumitru Potop-Butucaru. Predicate-aware, makespan-preserving software pipelining of scheduling tables. *TACO*, 11(1):12, 2014.

[38] Thomas Carle, Dumitru Potop-Butucaru, Yves Sorel, and David Lesens. From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. *LITES*, 2015. to appear.

[39] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):18, Sep 2001.

[40] P. Caspi, A. Curic, A. Magnan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings LCTES*, San Diego, CA, USA, June 2003.

[41] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, May/June 1999.

[42] Damien Chabrol, Vincent David, Christophe Aussaguès, Stéphane Louise, and Frédéric Daumas. Deterministic distributed safety-critical real-time systems within the oasis approach. In *Proceedings IASTED PDCS*, pages 260–268, 2005.

[43] Damien Chabrol, Didier Roux, Vincent David, Mathieu Jan, Moha Ait Hmid, Patrice Oudin, and Gilles Zeppa. Time- and angle-triggered real-time kernel. In *Proceedings DATE*, pages 1060–1062, 2013.

[44] D. Chapiro. *Globally-Asynchronous Locally- Synchronous Systems*. PhD thesis, Stanford University, 1984. Report No. STAN-CS-84-1026.

[45] K. Chatha and R. Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(3):193–208, 2002.

[46] Chunqing Chen, Jun Sun, Yang Liu, Jin Song Dong, and Manchun Zheng. Formal modeling and validation of stateflow diagrams. *STTT*, 14(6):653–671, 2012.

[47] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[48] Yi-Sheng Chiu, Chi-Sheng Shih, and Shih-Hao Hung. Pipeline schedule synthesis for real-time streaming tasks with inter/intra-instance precedence constraints. In *DATE*, Grenoble, France, 2011.

[49] A. Cohen, V. Perrelle, D. Potop-Butucaru, E. Soubiran, and Zhen Zhang. Mixed-criticality in railway systems: A case study on signaling application. In *Proceedings WMCIS 2015*, Paris, France, 2015.

[50] S. S. Craciunas and R. Serna Oliver. Smt-based task- and network-level static schedule generation for time-triggered networked systems. In *Proceedings RTNS*, Versailles, France, October 2014.

[51] A. Curic. *Implementing Lustre programs on distributed platforms with real-time constraints.* PhD thesis, Universite Joseph Fourier, Grenoble, 2005.

[52] R.I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), October 2011.

[53] Robert de Simone and Charles André. Time modeling in MARTE. In *Proceedings FDL 2007, September 18-20, 2007, Barcelona, Spain, Proceedings*, pages 268–273, 2007.

[54] V. Diekert and G. Rozenberg, editors. *The book of traces.* World Scientific, 1995.

[55] Manel Djemal, Robert de Simone, François Pêcheux, Franck Wajsbürt, Dumitru Potop-Butucaru, and Zhen Zhang. Programmable routers for efficient mapping of applications onto noc-based mpsocs. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing, DASIP 2012, Karlsruhe, Germany, October 23-25, 2012*, 2012.

[56] J. Doyle, B. Francis, and A. Tannenbaum. *Feedback Control Theory.* Macmillan Publishing Co., 1990.

[57] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th annual conference on Design automation.* SESSION: Wild and crazy ideas (WACI), June 2007.

[58] S.A. Edwards, S. Kim, E.A. Lee, I. Liu, H.D. Patel, and M. Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proceedings ICCD.* IEEE, October 2009. Lake Tahoe, CA.

[59] P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on VLSI Systems*, 8(5):472–491, Oct 2000.

[60] Embedded.com. 2009 embedded market study. Online, Jan 2009. `http://www.embedded.com/electronics-blogs/embedded-market-surveys/4405221/2009-Embedded-Market-Survey`.

[61] E. Waingold *et al.* Baring it all to software: The raw machine. *IEEE Computer*, 30(9):86 –93, sep 1997.

[62] J. Howard *et al.* A 48-core ia-32 processor in 45nm cmos using on-die message-passing and dvfs for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1), Jan 2011.

[63] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2), April 1985.

[64] Sebastian Fischmeister, Oleg Sokolsky, and Insup Lee. Network-code machine: Programmable real-time communication schedules. In *Proceedings RTAS*, pages 311–324, 2006.

[65] G. Fohler. Changing operational modes in the context of pre run-time scheduling, 1993.

[66] G. Fohler and K. Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *In Procs. of EUROMICRO-RTS97*, pages 128–135, 1995.

[67] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[68] F. Gasperoni and Uwe Schwiegelshohn. Generating close to optimum loop schedules on parallel processors. *Parallel Processing Letters*, 4(4):391–404, December 1994.

[69] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat. Time analysable synchronisation techniques for parallelised hard real-time applications. In *Proceedings DATE'12*, Dresden, Germany, 2012.

[70] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3), Sep 2003.

[71] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5), 2005.

[72] R. Gorcitz, E. Kofman, T. Carle, D. Potop-Butucaru, and R. de Simone. On the scalability of constraint solving for static/off-line real-time scheduling. In *Proceedings FORMATS 2015*, Madrid, Spain, 2015.

[73] R.A. Gorcitz, D. Monchaux, T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens. Automatic implementation of ttethernet-based time-triggered avionics applications. In *Proceedings DASIA 2015*, Barcelona, Spain, 2015.

[74] T. Goubier, R. Sirdey, S. Louise, and V. David. σc: A programming model and language for embedded manycores. In *Proceedings ICA3PP'11 (LNCS 7016)*, Melbourne, Australia, Oct 2011.

[75] R. Govindarajan, E. Altman, and G. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, 1994.

[76] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[77] Paul Le Guernic, Jean pierre Talpin, and Jean christophe Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12:261–304, 2002.

[78] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[79] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings ACSD*, pages 3–14, 2006.

[80] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[81] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[82] D. Harel and A. Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[83] M. Harrand and Y. Durand. Network on chip with quality of service. United States patent application publication US 2011/026400A1, Feb. 2011.

[84] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4^{th} edition, 2007.

[85] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[86] T.A. Henzinger and C. Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems*, 29(6), Oct 2007.

[87] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[88] C. Hilton and B. Nelson. PNoC: a flexible circuit-switched noc for fpga-based systems. *IEE Proceedings on Computers and Digital Techniques*, 153(3), 2006.

[89] R.A. Huff. Lifetime-sensitive modulo scheduling. In *In Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, 1993.

[90] IEEE. *IEEE Standard 1364-2005 for Verilog Hardware Description Language*, 2005.

[91] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[92] Mohammad Reza Kakoee. *Reliable and Variation-tolerant Interconnection Network for Low Power MPSoCs*. PhD thesis, Universitá di Bologna, 2012. Online at `http://amsdottorato.unibo.it/4407/1/phdthesis.pdf`.

[93] H. Kashif, S. Gholamian, R. Pellizzoni, H.D. Patel, and S. Fischmeister. Ortap: An offset-based response time analysis for a pipelined communication resource model. In *Proceedings RTAS*, 2013.

[94] O. Kermia and Y. Sorel. A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. In *Proceedings of ISCA 20th International Conference on Parallel and Distributed Computing Systems, PDCS'07*, Las Vegas, Nevada, USA, September 2007.

[95] Wonsub Kim, Donghoon Yoo, Haewoo Park, and Minwook Ahn. Scc based modulo scheduling for coarse-grained reconfigurable processors. In *Field-Programmable Technology (FPT), 2012 International Conference on*, Seoul, Korea, 2012.

[96] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *LNCS 563*, volume 563 of *Lecture Notes in Computer Science*, pages 87–101, 1991.

[97] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[98] M. Lam. Software pipelining : An effective scheduling technique for vliw machines. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[99] P. Le Guernic and A. Benveniste. Real-time, synchronous, data-flow programming: the language signal and its mathematical semantics. Research Report RR-620, INRIA, 1987.

[100] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[101] LIP6. SoClib: an open platform for virtual prototyping of multi-processors system on chip, 2011. Online at: http://www.soclib.fr.

[102] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 14, No. 2:46–61, january 1973.

[103] S. Louise, M. Lemerre, C. Aussagues, and V. David. The OASIS kernel: A framework for high dependability real-time systems. In *Proceedings of the 13th international symposium on High-Assurance Systems Engineering (HASE)*, Boca Raton, FL, USA, Nov 2011.

[104] Zhonghai Lu and A. Jantsch. Tdm virtual-circuit configuration for network-on-chip. *IEEE Trans. VLSI*, 2007.

[105] Nancy Lynch and Eugene Stark. A proof of the kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, 1989.

[106] M. Marouf, L. George, and Y. Sorel. Schedulability analysis for a combination of non-preemptive strict periodic tasks and preemptive sporadic tasks. In *Proceedings ETFA'12*, Kraków, Poland, September 2012.

[107] J.F. Mason, K. R. Luecke, and J.A. Luke. Device drivers in time and space partitioned operating systems. In *25th Digital Avionics Systems Conference, IEEE/AIAA*, Portland, OR, USA, Oct. 2006.

[108] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In *Proceedings DAC'12*, San Francisco, CA, USA, June 2012.

[109] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network-on-chip. In *Proceedings DATE*, 2004.

[110] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[111] L. Morel. *Exploitation des structures régulières et des spécifications locales pour le developpement correct de systèmes réactifs de grande taille*. PhD thesis, Institut National Polytechnique de Grenoble, 2005.

[112] T. Moscibroda and O. Mutlu. A case for bufferless routing in on-chip networks. In *Proceedings ISCA-36*, 2009.

[113] The MPPA256 many-core architecture. `www.kalray.eu`, 2012.

[114] L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2), 1993.

[115] B. Nikolic, H. Ali, S.M. Petters, and L.M. Pinho. Are virtual channels the bottleneck of priority-aware wormhole-switched noc-based many-cores? In *Proceedings RTNS*, 2013, October 2013.

[116] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.

[117] I. Miro Panades, A. Greiner, and A. Sheibanyrad. A low cost network-on-chip with guaranteed service well suited to the GALS approach. In *Proceedings NanoNet'06*, Lausanne, Switzerland, Sep 2006.

[118] V. Papailiopoulou, D. Potop-Butucaru, Y. Sorel, R. De Simone, L. Besnard, and J.-P. Talpin. From design-time concurrency to effective implemen- tation parallelism: The multi-clock reactive case. In *Proceedings ESLsyn 2011*, San Diego, CA, USA, 2011.

[119] P. Pop, P. Eles, and Z. Peng. Scheduling with optimized communication for time-triggered embedded systems. In *Proceedings CODES'99*, 1999.

[120] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Deterministic execution of synchronous programs in an asynchronous environment. a compositional necessary and sufficient condition. Research Report RR-6656, INRIA, September 2008. https://hal.inria.fr/inria-00322563.

[121] D. Potop-Butucaru, R. De Simone, and Y. Sorel. From synchronous specifications to statically-scheduled hard real-time implementations. In *S. Shukla, J.-P. Talpin (eds.), Synthesis of Embedded Software*. Springer, 2010. ISBN: 978-1-4419-6399-4.

[122] D. Potop-Butucaru and Y. Sorel. Synchronous approach and scheduling. In Wiley-ISTE, editor, *M. Chetto (ed.) Real-Time Systems Scheduling, vol. 2 Focuses.*, 2014.

[123] Dumitru Potop-Butucaru. The Kahn principle for networks of synchronous endochronous programs. In *Proceedings FMGALS 2003*, Pisa, Italy, Sep. 2003.

[124] Dumitru Potop-Butucaru, Akramul Azim, and Sebastian Fischmeister. Semantics-preserving implementation of synchronous specifications over dynamic TDMA distributed architectures. In *Proceedings of the 10th International conference on Embedded software, EMSOFT 2010, Scottsdale, Arizona, USA, October 24-29, 2010*, 2010.

[125] Dumitru Potop-Butucaru and Benoît Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France*, 2005.

[126] Dumitru Potop-Butucaru and Benoît Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundam. Inform.*, 78(1):131–159, 2007.

[127] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in synchronous systems. In *Proccedings of the 4th International Conference on Application of Concurrency to System Design (ACSD 2004), 16-18 June 2004, Hamilton, Canada*, 2004.

[128] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.

[129] Dumitru Potop-Butucaru, Robert de Simone, and Yves Sorel. Necessary and sufficient conditions for deterministic desynchronization. In *Proceedings of the 7th ACM & IEEE International conference on Embedded software, EMSOFT 2007, September 30 - October 3, 2007, Salzburg, Austria*, 2007.

[130] Dumitru Potop-Butucaru, Robert de Simone, Yves Sorel, and Jean-Pierre Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, 2009.

[131] Dumitru Potop-Butucaru, Robert de Simone, Yves Sorel, and Jean-Pierre Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. In *Proceedings of the Ninth International Conference on Application of Concurrency to System Design, ACSD 2009, Augsburg, Germany, 1-3 July 2009*, 2009.

[132] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, 2007.

[133] Dumitru Potop-Butucaru and Isabelle Puaut. Integrated worst-case execution time estimation of multicore applications. In *Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, 2013.

[134] Dumitru Potop-Butucaru, Yves Sorel, Robert de Simone, and Jean-Pierre Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. *Fundam. Inform.*, 108(1-2):91–118, 2011.

[135] C. Pradalier, J. Hermosillo, C. Koike, C. Braillon, P. Bessière, and C. Laugier. The CyCab: a car-like robot navigating autonomously and safely among pedestrians. *Robotics and Autonomous Systems*, 50(1), 2005.

[136] W. Puffitsch, E. Noulard, and C. Pagetti. Mapping a multi-rate synchronous language to a many-core processor. In *Proceedings RTAS*, 2013.

[137] R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7(3), May 2008.

[138] A. Racu and L.S. Indrusiak. Using genetic algorithms to map hard real-time on noc-based systems. In *Proceedings ReCoSoC*, July 2012.

[139] K. Ramamritham, G. Fohler, and J. M. Adan. Issues in the static allocation and scheduling of complex periodic tasks. In *In Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software*, 1993.

[140] B.R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1):3–64, 1996.

[141] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming, IEEE*, 1981.

[142] J. Rushby. Bus architectures for safety-critical embedded systems. In *Proceedings EMSOFT'01*, volume 2211 of *LNCS*, Tahoe City, CA, USA, 2001.

[143] A. Al Sheikh, O. Brun, P.-E. Hladik, and B.J. Prabhu. Strictly periodic scheduling in ima-based architectures. *Real-Time Systems*, 48(4):359–386, 2012.

[144] Z. Shi and A. Burns. Schedulability analysis and task mapping for real-time on-chip communication. *Real-Time Systems*, 46(3):360–385, 2010.

[145] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Proceedings DATE'04*, Paris, France, 2004.

[146] M. Smelyanskyi, S. Mahlke, E. Davidson, and H.-H. Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *Proceedings CGO*, San Francisco, CA, USA, March 2003.

[147] R.B. Sorensen, M. Schoeberl, and J. Sparso. A light-weight statically scheduled network-on-chip. In *Proceedings NORCHIP*, 2012.

[148] The TilePro64 many-core architecture. `www.tilera.com`, 2008.

[149] M. Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings Memocode'09*, pages 171–180, Nice, France, 2009.

[150] C.Y. Villalpando, A.E. Johnson, R. Some, J. Oberlin, and S. Goldberg. Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander. In *Proceedings of the IEEE Aerospace Conference*, 2010.

[151] VSI Alliance. VCI: Virtual Component Interface Standard (OCB 2 2.0). Online at: http://www.vsi.org.

[152] J. Wang and Christine Eisenbeis. Decomposed software pipelining. `http://hal.inria.fr/inria-00074834`, 1993.

[153] N.J. Warter, D. M. Lavery, and W.W. Hwu. The benefit of predicated execution for software pipelining. In *HICSS-26 Conference Proceedings*, Houston, Texas, USA, 1993.

[154] R. Wilhelm and J. Reineke. Embedded systems: Many cores – many problems (invited paper). In *Proceedings SIES'12*, Karlsruhe, Germany, June 2012.

[155] Jia Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *Software Engineering, IEEE Transactions on*, 19(2):139–154, 1993.

[156] Hoeseok Yang and Soonhoi Ha. Pipelined data parallel task mapping/scheduling technique for mpsoc. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, Nice, France, 2009.

[157] Y.J. Yoon, N. Concer, M. Petracca, and L. Carloni. Virtual channels vs. multiple physical networks: a comparative analysis. In *Proceedings DAC*, Anaheim, CA, USA, 2010.

[158] H.-S. Yun, J. Kim, and S.-M. Moon. Time optimal software pipelining of loops with control flows. *International Journal of Parallel Programming*, 31(5):339–391, October 2003.

[159] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Register constrained modulo scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 15(5):417–430, 2004.

[160] J. T. Zhai, M. Bamakhrama, and T. Stefanov. Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *Proceedings DAC*, 2013.

[161] W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Extensible and scalable time-triggered scheduling. In *Proceedings ACSD*, St. Malo, France, June 2005.

[162] Q. Zhuge, Z. Shao, and E.H. Sha. Optimal code size reduction for software-pipelined loops on dsp applications. In *Proceedings of the International Conference on Parallel Processing*, 2002.